Volume 15 Number 2

ACTA CYBERNETICA

Editor-in-Chief: J. Csirik (Hungary)

Managing Editor: Z. Fülöp (Hungary)

Assistants to the Managing Editor: A. Pluhár (Hungary), B. Tóth (Hungary)

Editors: M. Arató (Hungary), S. L. Bloom (USA), H. L. Bodlaender (The Netherlands), W. Brauer (Germany), L. Budach (Germany), H. Bunke (Switzerland), B. Courcelle (France), J. Demetrovics (Hungary), B. Dömölki (Hungary), J. Engelfriet (The Netherlands), Z. Ésik (Hungary), F. Gécseg (Hungary), J. Gruska (Slovakia), B. Imreh (Hungary), H. Jürgensen (Canada), A. Kelemenová (Czech Republic), L. Lovász (Hungary), G. Păun (Romania), A. Prékopa (Hungary), A. Salomaa (Finland), L. Varga (Hungary), H. Vogler (Germany), G. Wöginger (Austria)

Szeged, 2001

ACTA CYBERNETICA

Information for authors. Acta Cybernetica publishes only original papers in the field of Computer Science. Contributions are accepted for review with the understanding that the same work has not been published elsewhere.

Manuscripts must be in English and should be sent in triplicate to any of the Editors. On the first page, the *title* of the paper, the *name(s)* and *affiliation(s)*, together with the *mailing* and *electronic address(es)* of the author(s) must appear. An *abstract* summarizing the results of the paper is also required. References should be listed in alphabetical order at the end of the paper in the form which can be seen in any article already published in the journal. Manuscripts are expected to be made with a great care. If typewritten, they should be typed double-spaced on one side of each sheet. Authors are encouraged to use any available dialect of TEX.

After acceptance, the authors will be asked to send the manuscript's source TEX file, if any, on a diskette to the Managing Editor. Having the TEX file of the paper can speed up the process of the publication considerably. Authors of accepted contributions may be asked to send the original drawings or computer outputs of figures appearing in the paper. In order to make a photographic reproduction possible, drawings of such figures should be on separate sheets, in India ink, and carefully lettered.

There are no page charges. Fifty reprints are supplied for each article published.

Publication information. Acta Cybernetica (ISSN 0324–721X) is published by the Department of Informatics of the University of Szeged, Szeged, Hungary. Each volume consists of four issues, two issues are published in a calendar year. For 2001 Numbers 1–2 of Volume 15 are scheduled. Subscription prices are available upon request from the publisher. Issues are sent normally by surface mail except to overseas countries where air delivery is ensured. Claims for missing issues are accepted within six months of our publication date. Please address all requests for subscription information to: Department of Informatics, University of Szeged, H-6701 Szeged, P.O.Box 652, Hungary. Tel.: (36)-(62)-420-184, Fax:(36)-(62)-420-292.

URL access. All these information and the contents of the last some issues are available in the Acta Cybernetica home page at http://www.inf.u-szeged.hu/local/acta.

EDITORIAL BOARD

Editor-in-Chief: **J. Csirik**
University of Szeged
Department of Computer Science
Szeged, Árpád tér 2.
H-6720 Hungary

Managing Editor: **Z. Fülöp**
University of Szeged
Department of Computer Science
Szeged, Árpád tér 2.
H-6720 Hungary

Assistants to the Managing Editor:

A. Pluhár
University of Szeged
Department of Computer Science
Szeged, Árpád tér 2.
H-6720 Hungary

B. Tóth
University of Szeged
Department of Computer Science
Szeged, Árpád tér 2.
H-6720 Hungary

Editors:

M. Arató
University of Debrecen
Department of Mathematics
Debrecen, P.O. Box 12
H-4010 Hungary

S. L. Bloom
Stevens Intitute of Technology
Department of Pure and Applied
Mathematics Castle Point, Hoboken
New Jersey 07030, USA

H. L. Bodlaender
Department of Computer Science
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

W. Brauer
Institut für Informatik
Technische Universität München
D-80290 München
Germany

L. Budach
University of Postdam
Department of Computer Science
Am Neuen Palais 10
14415 Postdam, Germany

F. Gécseg
University of Szeged
Department of Computer Science
Szeged, Aradi vértanúk tere 1.
H-6720 Hungary

J. Gruska
Institute of Informatics/Mathematics
Slovak Academy of Science
Dúbravska 9, Bratislava 84235
Slovakia

B. Imreh
University of Szeged
Department of Foundations of
Computer Science
Szeged, Aradi vértanúk tere 1.
H-6720 Hungary

H. Jürgensen
The University of Western Ontario
Department of Computer Science
Middlesex College, London, Ontario
Canada N6A 5B7

A. Kelemenová
Institute of Mathematics and
Computer Science
Silesian University at Opava
761 01 Opava, Czech Republic

H. Bunke
Universität Bern
Institut für Informatik und
angewandte Mathematik
Längass strasse 51., CH-3012 Bern
Switzerland

B. Courcelle
Universitè Bordeaux-1
LaBRI, 351 Cours de la Libèration
33405 TALENCE Cedex
France

J. Demetrovics
MTA SZTAKI
Budapest, Lágymányosi u. 11
H-1111 Hungary

B. Dömölki
IQSOFT
Budapest, Teleki Blanka u. 15-17.
H-1142 Hungary

J. Engelfriet
Leiden University
LIACS
P.O. Box 9512, 2300 RA Leiden
The Netherlands

Z. Ésik
University of Szeged
Department of Foundations of
Computer Science
Szeged, Aradi vértanúk tere 1.
H-6720 Hungary

L. Lovász
Eötvös Loránd University
Budapest, Kecskeméti u. 10-12.
H-1053 Hungary

G. Păun
Institute of Mathematics
Romanian Academy
P.O.Box 1-764, RO-70700
Bucuresti, Romania

A. Prékopa
Eötvös Loránd University
Budapest, Kecskeméti u. 10-12.
H-1053 Hungary

A. Salomaa
University of Turku
Department of Mathematics
SF-20500 Turku 50, Finland

L. Varga
Eötvös Loránd University
Budapest, Pázmány Péter sétány 1/c.
H-1117 Hungary

H. Vogler
Dresden University of Technology
Department of Computer Science
Foundations of Programming
D-01062 Dresden, Germany

G. Wöginger
Department of Matematics
University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherland

Preface

The Second Conference for PhD Students in Computer Science (CSCS) was organized by the Department of Computer Science of the University of Szeged (SZTE) and held in Szeged, Hungary from July 20 to 23, 2000. The members of the Scientific Committee were the following representants of the Hungarian doctoral schools in computer science: Mátyás Arató (KLTE), András Benczúr (ELTE), Miklós Bartha (SZTE), Tibor Csendes (SZTE), János Csirik (SZTE), János Demetrovics (SZTAKI), Sarolta Dibuz (Ericsson), József Dombi (SZTE), Zoltán Ésik (SZTE), Ferenc Friedler (VE), Zoltán Fülöp (SZTE), Ferenc Gécseg (chair, SZTE), Balázs Imreh (SZTE), János Kormos (KLTE), László Kozma (ELTE), Attila Kuba (SZTE), Eörs Máté (SZTE), Gyula Pap (KLTE), András Recski (BME), Endre Selényi (BME), Katalin Tarnay (NOKIA), György Turán (JATE), and László Varga (ELTE). The members of the Organizing Committee were Tibor Csendes, Éva Gombás, Mihály Csaba Markót, László Martonossy, Lajos Schrettner, and Mariann Sebő.

There were more than 80 participants and 60 talks in several fields of computer science and its applications. Beyond the Hungarian PhD schools in computer science, mainly the universities of Almería, Spain and of Nis, Yugoslavia were represented. The talks were going in two parallel sections in artificial intelligence, automata and formal languages, computer networks, database theory, discrete mathematics, formal languages, fuzzy decision support systems, information systems, optimization, picture processing, and software engineering. The talks of the students were completed by 4 plenary talks of leading scientists.

Three scientific journals, viz. Periodica Polytechnica (Budapest), Publicationes Mathematicae (Debrecen) and Acta Cybernetica (Szeged) offered students to publish the paper version of their presentations after a selection and review process. Altogether 38 papers were submitted for publication. The present special issue of Acta Cybernetica contains 14 such papers, and 4 further will be published in a later regular issue.

The full program of the conference, the collection of the abstracts and further information can be found at `http://www.inf.u-szeged.hu/~cscs`.

On the basis of our positive experiences, the conference will be organized in the future, too, hopefully with more foreign participants. According to the present plans, the next meeting will be held in July 2002 in Szeged.

Tibor Csendes and Zoltán Fülöp

Acta Cybernetica 15 (2001) 121–137.

Sets of integers in different number systems and the Chomsky hierarchy

István Katsányi *

Abstract

The classes of the Chomsky hierarchy are characterized in respect of converting between canonical number systems. We show that the relations of the bases of the original and converted number systems fall into four distinct categories, and we examine the four Chomsky classes in each of the four cases. We also prove that all of the Chomsky classes are closed under constant addition and multiplication. The classes $\mathcal{RE}$ and $\mathcal{CS}$ are closed under every examined operation. The regular languages are closed under addition, but not under multiplication.

Key words: formal languages, Chomsky hierarchy, canonical number systems, number theory, converting.

1 Introduction

It is a thoroughly studied subject within the discipline of formal languages and automata theory, that under which conditions will a set of integers in m–ary notation be regular for a given $m \geq 1$. Cobham has solved the bases of this problem in [1]. His results were extended and generalized by many authors in many ways for example in the papers [2], [3], [4], [5], [6], [7]. Luca and Restivo suggested in their paper ([3]) to study the open problem of the context–free case. In this work, we examine the context–free, the context sensitive and the recursively enumerable classes in addition to the regular languages, hence the examination of the Chomsky–hierarchy in this regard becomes complete.

We also investigate, that under what conditions do certain arithmetical operations alter the Chomsky–class of a language. We prove some closure properties and generalize a theorem related to ranges of polynomials.

2 Preliminaries

When not stated otherwise, we will use the standard notations used in the theory of formal languages (see for example [8]). The set of nonnegative integers will

*Eötvös Loránd University, Department of General Computer Science, 1117 Budapest, Pázmány Péter sétány 1/D, e-mail: kacsa@ludens.elte.hu

be denoted by $\mathcal{N}$, and the classes of regular, context–free, context sensitive and recursively enumerable languages by $\mathcal{REG}$, $\mathcal{CF}$, $\mathcal{CS}$, $\mathcal{RE}$, respectively. The length of a word u will be denoted by $|u|$. The notation of the empty word is λ. The mirror image of a word u will be denoted by u^{-1}. We use the same notation for the mirror images of languages: for a language L let $L^{-1} = \{u^{-1} \mid u \in L\}$.

We call a word u a proper base–a integer ($a \geq 1$), if $a = 1$ and u consists of some 1 digits, or $a \geq 2$, u consists of digits $0, \ldots, a-1$, and $u = 0$, or the first digit of u is nonzero. The value of a proper base–a integer u will be denoted by $val_a(u)$. (The value of a proper base–1 integer u is the length of u.) We denote by $L_a(A)$ the language of proper base–a integers, whose values constitutes the set A, where A is a set of nonnegative integers, and a is a positive integer. For example, if $A = \{2^n \mid n \geq 0\}$, then $L_2(A) = \{1, 10, 100, \ldots\}$ can be expressed by the regular expression 10^*, but according to Büchi ([9]) $L_{10}(A) = \{1, 2, 4, 8, 16, \ldots\}$ and $L_1(A) = \{1, 11, 1111, \ldots\}$ are nonregular languages.

We call two integers $a, b \geq 2$ *multiplicatively dependent*, if there exist integers $n, m \geq 1$, such that $a^n = b^m$. Otherwise, they are called *multiplicatively independent.*

We will also use the well–known structure of generalized sequential machines, and a pumping lemma concerning regular languages ([10],[11], [12]):

Definition 1 *A* generalized sequential machine *is a* 6*-tuple* $(Q, \Sigma, \Delta, \sigma, s, F)$, *where* Q *is the finite set of states,* Σ *is the input alphabet,* Δ *is the output alphabet,* σ *is the transition–and–output function from* $Q \times \Sigma$ *to finite subsets of* $Q \times \Delta^*$, $s \in Q$ *is the starting state, and* $F \subseteq Q$ *is the set of final states. For a gsm* G, *the set of all output words in response to an input word* $u \in \Sigma^*$ *is denoted by* $G(u)$. *For a language* $L \subseteq \Sigma^*$ *we define* $G(L) = \bigcup_{u \in L} G(u)$.

Proposition 1 *Let* L *be a regular set. Then there is a constant* n, *such that if* z *is any word in* L, *and* $|z| \geq n$, *we may write* $z = uvw$ *in such a way that* $|uv| \leq n$, $|v| \geq 1$, *and for all* $i \geq 0$, uv^iw *is in* L.

3 Converting between canonical number systems

One of the main results of our paper is the next theorem:

Theorem 1

	$a = 1, b \geq 2$	$a \geq 2, b = 1$	$a, b \geq 2$, $\exists n, m \geq 1 :$ $a^n = b^m$	$a, b \geq 2$, $\nexists n, m \geq 1 :$ $a^n = b^m$
$\mathcal{REG}$	$\mathcal{REG}$	$\mathcal{CS}$	$\mathcal{REG}$	$\mathcal{CS}^*$
$\mathcal{CF}$	$\mathcal{REG}$	$\mathcal{CS}$	$\mathcal{CF}$	$\mathcal{CS}^*$
$\mathcal{CS}$	$\mathcal{RE}^*$	$\mathcal{CS}$	$\mathcal{CS}$	$\mathcal{CS}$
$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$

*For each of the 16 inner cells of the table the following holds: If a and b are integers that satisfy the condition shown in the heading of the column of the cell, and A is a set of integers such that $L_a(A)$ belongs to the class shown in the heading of the row of the cell, then the language $L_b(A)$ belongs to the class shown in the cell. For each cell that is not marked with a *, there exists integers a, b and sets A, such that they satisfy the appropriate conditions, and $L_b(A)$ is not the element of any smaller Chomsky class, than the one shown in the cell.*

The theorem can be proved by a series of lemmas. Let us prove a part of the context–sensitive case first.

Lemma 1 *Let $a \geq 2$ be an integer, and let A be a set of nonnegative integers, such that the language $L_a(A)$ is context–sensitive. Then for all $b \geq 1$ integers, the language $L_b(A)$ is also context–sensitive.*

Proof. We will construct a grammar $G' = (N', T', S', P')$, such that $L(G') = L_b(A)$ and then we will prove, that there is a constant c such that there exists a derivation in G' of every nonempty word $u \in L(G')$, which uses a workspace of size $c|u|$ or less. For the proof, that this property is sufficient for the generated language to be context sensitive, see e.g. [8], Theorem 10.1.

Let $G = (N, T, S, P)$ be a length–increasing grammar generating the language $L_a(A)$. We may assume, that $T = \{0, 1, \ldots, a-1\}$, and the left–side of each production of G contains only nonterminals, and even the right–sides of the productions contain only terminals in productions of the form $X_i \to i$ ($X_i \in N, 0 \leq i \leq a-1$), and these X_i nonterminals do not appear on the left side of other productions. G' works roughly as follows: it generate words according to the rules of G, enclose it between some markers, and creates a separate block, where the base–b representation of the generated number will evolve. This block first consists of the representation of 0, and then the rules of G' one by one decrease the generated base–a number, and increase the base–b number in the separate block.

Let us suppose first, that $b \geq 2$. Then G' will be the following:

$$N' = N \cup \{S', B, M, E, C, D, F, H, Y_0, Y_1, \ldots, Y_{b-1}\},$$
$$T' = \{0, 1, \ldots, b-1\},$$
$$P' = P \cup P'',$$

where the newly added nonterminals do not appear in N, and P'' consists of the following rules:

Beginning:

$$S' \to BSCMY_0E \tag{1}$$

Subtracting 1 from the left side:

$$X_iX_jC \to X_iX_{j-1}D \qquad 0 \le i \le a-1 \quad 1 \le j \le a-1 \qquad (2)$$
$$X_iX_0C \to X_iCX_{a-1} \qquad 0 \le i \le a-1 \qquad (3)$$
$$BX_iC \to BX_{i-1}D \qquad 2 \le i \le a-1 \qquad (4)$$
$$BX_1C \to BD \qquad (5)$$

D goes to the right:

$$DX_i \to X_iD \qquad 0 \le i \le a-1 \qquad (6)$$
$$DM \to MD \qquad (7)$$
$$DY_i \to Y_iD \qquad 0 \le i \le b-1 \qquad (8)$$

Adding 1 to the right side:

$$DE \to FE \qquad (9)$$
$$Y_iF \to HY_{i+1} \qquad 0 \le i \le b-2 \qquad (10)$$
$$Y_{b-1}F \to FY_0 \qquad (11)$$
$$MF \to MHY_1 \qquad (12)$$

H goes to the left:

$$Y_iH \to HY_i \qquad 0 \le i \le b-1 \qquad (13)$$
$$MH \to CM \qquad (14)$$

If the left side is empty, we have finished:

$$BX_0CM \to BCM \qquad (15)$$
$$BCM \to \lambda \qquad (16)$$
$$E \to \lambda \qquad (17)$$
$$Y_i \to i \qquad 0 \le i \le b-1 \qquad (18)$$

In order to prove that $L_b(A) = L(G')$ first we prove, that $L_b(A) \subseteq L(G')$. Let us prove two claims first, which shows that the encoded numbers between the symbols B and M (M and E) can be decreased (increased) by one, under certain conditions. Using these two claims the proof of the inclusion will be easy.

Claim 1 *Let u be a word of the form*

$$u = BX_{j_1} \dots X_{j_k}CMu_0E,$$

where $k \ge 1$, $j_1 \dots j_k$ is a proper base-a integer of a positive value, and $u_0 \in \{Y_0, \dots, Y_{b-1}\}^*$. Then

$$u \Longrightarrow^*_{G'} u' = BX_{j'_1} \dots X_{j'_{k'}}DMu_0E,$$

such that either $k = 1$, $X_1 = 1$ *and* $k' = 0$, *or* $j'_1 \dots j'_{k'}$ *is a proper base–a integer, and* $val_a(j_1 \dots j_k) = val_a(j'_1 \dots j'_{k'}) + 1$.

Proof. Using rules of type (3) the symbol C steps over all X_0 nonterminals, and replace each of them by X_{a-1}. (We cannot decrease the digit zero any more, we must replace it by the maximal digit and try to decrease the preceding digit.)

$$u \Longrightarrow^*_{G'} u_1 = BX_{j_1} \dots X_{j_m} C \overbrace{X_{a-1} \dots X_{a-1}}^{k-m} M u_0 E,$$

where $1 \leq m \leq k$ and $j_m \geq 1$.
If m is at least two, we may „subtract one" form X_{j_m} using a rule of type (2):

$$u_1 \Longrightarrow_{G'} u_2 = BX_{j_1} \dots X_{j_m - 1} D \overbrace{X_{a-1} \dots X_{a-1}}^{k-m} M u_0 E,$$

If m is one, but $j_1 \geq 2$, we can still „decrease" X_{j_1} using a rule of type (4):

$$u_1 \Longrightarrow_{G'} u_3 = BX_{j_1 - 1} D \overbrace{X_{a-1} \dots X_{a-1}}^{k-1} M u_0 E,$$

Finally, if both m and j_1 equal to one, we erase X_1 using rule (5):

$$u_1 \Longrightarrow_{G'} u_4 = BD \overbrace{X_{a-1} \dots X_{a-1}}^{k-1} M u_0 E,$$

From each of u_2, u_3 and u_4 we can derive u' by using rules of type (6). It follows from the construction, that the condition given for u' holds.

Claim 2 *Let* u *be a word of the form*

$$u = Bu_0 D M Y_{l_1} \dots Y_{l_s} E,$$

where $u_0 \in \{X_0, \dots, X_{a-1}\}^*$, $s \geq 1$ *and* $l_1 \dots l_s$ *is a proper base–b integer. Then*

$$u \Longrightarrow^*_{G'} u' = Bu_0 C M Y_{l'_1} \dots Y_{l'_{s'}} E,$$

where $s' \geq 1$, $l'_1 \dots l'_{s'}$ *is a proper base–b integer and*

$$val_b(l_1 \dots l_s) + 1 = val_b(l'_1 \dots l'_{s'}).$$

Proof. By rules of type (7), (8) and (9) we derive u_1:

$$u \Longrightarrow^*_{G'} u_1 = Bu_0 M Y_{l_1} \dots Y_{l_s} F E.$$

Then by the rule (11) F steps over all Y_{b-1}, which cannot be increased any more, so these are replaced by Y_0-s:

$$u_1 \Longrightarrow^*_{G'} u_2 = Bu_0 M Y_{l_1} \dots Y_{l_m} F \overbrace{Y_0 \dots Y_0}^{s-m} E,$$

where either $m = 0$ or $1 \leq m \leq s$ and $0 \leq l_m \leq b-2$. If m is at least 1, then we may apply a rule of type (10), and increase the rightmost non–maximal digit:

$$u_2 \Longrightarrow_{G'} u_3 = Bu_0MY_{l_1} \dots Y_{l_{m-1}} HY_{l_m+1} \overbrace{Y_0 \dots Y_0}^{s-m} E.$$

If m is zero, we use rule (12) and introduce a new digit:

$$u_2 \Longrightarrow_{G'} u_4 = Bu_0MHY_1 \overbrace{Y_0 \dots Y_0}^{s} E.$$

From both u_3 and u_4 we can derive u' by using rules of type (13) and (14). It follows from the construction, that the condition given for u' holds.

Now let us prove, that for every $v \in L_b(A)$ there exists a derivation

$$S' \Longrightarrow^*_{G'} v.$$

By definition, there must be a word $u \in L(G)$, such that $val_a(u) = val_b(v)$. Let us denote the digits of u by $i_1, i_2, \dots, i_{|u|}$. Hence $u = i_1 i_2 \dots i_{|u|}$ $(0 \leq i_1, i_2, \dots, i_{|u|} \leq a-1)$, and the following derivations are valid:

$$S \Longrightarrow^*_G X_{i_1} X_{i_2} \dots X_{i_{|u|}},$$
$$S' \Longrightarrow^*_{G'} BX_{i_1} X_{i_2} \dots X_{i_{|u|}} CMY_0E.$$

If $u = 0$, then we continue the derivation by the rules (15), (16), (17), and $Y_0 \to 0$. We derived 0, which is the base–b representation of u. If $u \neq 0$, then we may use Claim 1 and Claim 2 consecutively $val_a(u)$ times. By that

$$S' \Longrightarrow^*_{G'} BCMY_{j_1} \dots Y_{j_s} E,$$

such that $val_b(j_1 \dots j_s) = val_a(u)$. Using rules of type (16), (17), and (18) we obtain, that

$$S' \Longrightarrow^*_{G'} j_1 \dots j_s = v,$$

which is exactly what we wanted to show.

Now let us prove, that $L(G') \subseteq L_b(A)$. Let u be an arbitrary word in $L(G')$, and let its derivation be the following:

$$S' \Longrightarrow_{G'} BSCMY_0E \Longrightarrow^*_{G'} u \in T'^*$$

We must not use rules of type $X_i \to i$ $(0 \leq i \leq a-1)$ during the derivation, or else the nonterminals B and M cannot be erased because of the terminal i between them. As no X_i $(0 \leq i \leq a-1)$ nonterminals exist on the left side of a production

of another type in P', the steps of this derivation can be interchanged in such a way, that we get the following derivation:

$$S' \Longrightarrow^*_{G'} BX_{i_1} \dots X_{i_n} CMY_0E \Longrightarrow^*_{G'} \tag{19}$$
$$\Longrightarrow^*_{G'} BCMY_{j_1} \dots Y_{i_k} E \Longrightarrow_{G'} \tag{20}$$
$$\Longrightarrow_{G'} Y_{j_1} \dots Y_{i_k} E \Longrightarrow_{G'} \tag{21}$$
$$\Longrightarrow_{G'} Y_{j_1} \dots Y_{i_k} \Longrightarrow^*_{G'} \tag{22}$$
$$\Longrightarrow^*_{G'} j_1 \dots j_k = u \tag{23}$$

At each step of the derivation (20), there can be used only one rule. For that, and for the arguments mentioned before $val_a(i_1 \dots i_n) = val_b(j_1 \dots j_k)$. On the other hand obviously $S \Longrightarrow^*_G i_1 \dots i_n$, so $val_a(i_1 \dots i_n) \in A$, which means, that $u \in L_b(A)$.

Finally, let us prove, that $L(G') \in CS$. We will show, that for an appropriately chosen constant c every derivation of every word $u \in L(G')$ uses a workspace of size $c|u|$ or less. This condition is obviously stronger than required for $L(G')$ being context–sensitive.

Let us consider an arbitrary derivation $\mathcal{D} : S' \Longrightarrow^*_{G'} u$ of an arbitrary word $u \in L(G')$. It is obvious, that every derivation of the word $u = 0$ uses a workspace of size $6 = 6\,|u|$. Otherwise, we can split the derivation into two parts:

$$\mathcal{D} : S' \Longrightarrow^*_{G'} BCMu' \Longrightarrow^*_{G'} u.$$

We call the *first part* of the derivation $\mathcal{D}$ the steps, where the derived word contains the nonterminals B and M, and there are at least one symbol between them, not counting the symbols C and D. We will refer the rest of the derivation as the *second part.*

If we denote by v the base–a representation of $val_b(u)$, then the number of the letters between the nonterminals B and M is at most $|v| + 1$, because the grammar G is length–increasing. The one extra symbol could be C or D.

During the first part of the derivation there cannot be more than $|u|+1$ symbols between the letters M and E, and during the second part, each derived word is not longer than $|u| + 5$. Now the extra symbols are B, C, M, E, and one of the letters D, F and H. We obtained, that during the derivation $\mathcal{D}$ the length of the longest derived word is at most $|u| + |v| + 6$.

Since it can be shown, that $|v| \leq (1 + \frac{\log b}{\log a})\,|u|$, there must be a constant c, such that every derived word during the derivation D is shorter, than $c\,|u|$. This proves, that the language $L(G')$ is context–sensitive.

The case of $b = 1$ is similar, we only have to make some minor modification in the definition of the grammar and in the proofs. The main difference is in the rule sets (9)–(12), but the modification should cause the reader no trouble. □

The following lemma is a consequence of Church's thesis, but we will give a direct proof, too.

Lemma 2 *Let $a \geq 1$ be an integer, and let A be a set of nonnegative integers, such that the language $L_a(A)$ is recursively enumerable. Then for all $b \geq 1$ integers, the language $L_b(A)$ is also recursively enumerable.*

Proof. For the case of $a \geq 2$ we use the very same construction as the one used in Lemma 1, but this time we can not suppose the original grammar to be length–increasing, hence the resulting grammar may generate languages, which are not context–sensitive. For the unary case we also use the same ideas, but we change the rules (2)–(5) to the following one: $X_1C \to D$. Using the same thoughts, we may prove, that the language generated by the constructed grammar equals to $L_b(A)$.□

The following lemma is an immediate consequence of Lemma 2.

Lemma 3 *Let A be a set of nonnegative integers, such that the language $L_1(A)$ is context–sensitive. Then for all $b \geq 2$ integers, the language $L_b(A)$ is recursively enumerable.*

There are no known examples of sets, that have context–sensitive representation in the unary number system, and a non–context–sensitive representation in an other number system, so we cannot be sure, that the recursively enumerable class is the smallest one, which contains this type of languages. In contrast to this, we have nice results concerning multiplicatively dependent bases.

Lemma 4 *Let $a, b \geq 2$ be two multiplicatively dependent integers, and let A be a set of nonnegative integers. Then $L_a(A) \in \mathcal{F}$ holds iff $L_b(A) \in \mathcal{F}$ is true, where $\mathcal{F}$ is one of the classes $\mathcal{RE}$, $\mathcal{CF}$, $\mathcal{REG}$.*

Proof. We will construct a generalized sequential machine G, such that when the input of G is the mirror image of an integer expressed in the base–a number system, then the output of G is the mirror image of the same integer expressed in the base–b number system. This completes the proof, because the classes in question are all closed under gsm–mappings and mirror images.

Let n and m be two positive integers, such that $a^n = b^m$. (These numbers must exists, because a and b are multiplicatively dependent integers.) The constructed gsm is the following one:

$$G = (\{f\} \cup \{s_{i,j} \mid 0 \leq i \leq a^n - 1, 0 \leq j \leq n-1\}, T_a, T_b, \sigma, s_{0,0}, \{f\}),$$

where $T_a = \{0, 1, \ldots, a-1\}$, $T_b = \{0, 1, \ldots, b-1\}$, and σ is defined by the following. If n is at least two, then for all integers i, j and letter $x \in T_a$, such that $0 \leq i \leq a^{n-2} - 1$, and $0 \leq j \leq n-2$ we have

$$\sigma(s_{i,j}, x) = \{(s_{i+xa^j, j+1}, \lambda), (f, u)\},$$

where u is a word, such that u^{-1} is a proper base–b integer, and $val_b(u^{-1}) = i + xa^j$. Moreover (for any values of n), we have

$$\sigma(s_{i,n-1}, x) = \{(s_{0,0}, u0^{m-|u|}), (f, u)\},$$

where $0 \leq i \leq a^{n-1} - 1$, $x \in T_a$, and u is a word, such that u^{-1} is a proper base–b integer, and $val_b(u^{-1}) = i + xa^{n-1}$.

The work of the gsm G is based upon the fact, that we can divide the number to be converted to distinct n–digit blocks, and we can make the conversion in each block independently of the other blocks. Each n–digit block will be converted to an m–digit block in the base–b number system. The gsm reads n consecutive digits from its input, keeping the value of the read (reversed) number in its internal state. The read number must be less, then a^n, so the number of internal states will not be infinite. After reading n digits, G outputs m digits: the reverse of the base–b representation of the read number, using leading zeroes, if necessary, and then it resets its counter, and begins a new cycle. At any moment, G may stop its operation by writing the mirror image of the base–b representation of the stored number to the output. This time we omit leading zeroes. After that step, G goes to the unique final state, which has no following state, so this step can only be used with success at the end of the input. The details of the proof, that for every proper base–a integer u, $G(u^{-1})$ consists of exactly one word, and that $v = G(u^{-1})^{-1}$ is a proper base–b integer, such that $val_a(u) = val_b(v)$ are left to the reader. □

The following lemma is an immediate consequence of Lemma 1:

Lemma 5 *Let $a, b \geq 2$ be two multiplicatively independent integers, and let $A \subseteq \mathcal{N}$ be a set, such that $L_a(A) \in \mathcal{REG}$, or $L_a(A) \in \mathcal{CF}$. Then $L_b(A) \in \mathcal{CS}$.*

By the work of Büchi ([9]) we know, that if $a, b \geq 2$ are two multiplicatively independent integers and $A \subseteq \mathcal{N}$ is a set, such that $L_a(A) \in \mathcal{REG}$, then $L_b(A)$ may be nonregular. However, it is not known, that the language $L_b(A)$ can be non–context–free, too. Our conjunction is, that it can.

The following lemma is another consequence of Lemma 1. It follows from Büchi's Theorem, that this lemma cannot be further strengthened.

Lemma 6 *Let $a \geq 2$ be an integer and let $A \subseteq \mathcal{N}$ be a set, such that $L_a(A) \in \mathcal{REG}$, or $L_a(A) \in \mathcal{CF}$. Then $L_1(A) \in \mathcal{CS}$.*

Using the fact, that every context–free language over a one–letter alphabet is also regular (see for example [8], Theorem 7.3), we get the following consequence of Cobham's Theorem ([1]):

Lemma 7 *Let $b \geq 2$ be an integer and let $A \subseteq \mathcal{N}$ be a set, such that $L_1(A) \in \mathcal{REG}$, or $L_1(A) \in \mathcal{CF}$. Then $L_b(A) \in \mathcal{REG}$.*

At the end of this section we repeat the table of Theorem 1, indicating in each cell the number of the lemma, which proves the part of the theorem that belongs to the cell.

	$a=1, b\geq 2$	$a\geq 2, b=1$	$a,b\geq 2$, $\exists n,m\geq 1: a^n=b^m$	$a,b\geq 2$, $\nexists n,m\geq 1: a^n=b^m$
$\mathcal{REG}$	$\mathcal{REG}$ (7.)	$\mathcal{CS}$ (6.)	$\mathcal{REG}$ (4.)	$\mathcal{CS}^*$ (5.)
$\mathcal{CF}$	$\mathcal{REG}$ (7.)	$\mathcal{CS}$ (6.)	$\mathcal{CF}$ (4.)	$\mathcal{CS}^*$ (5.)
$\mathcal{CS}$	$\mathcal{RE}^*$ (3.)	$\mathcal{CS}$ (1.)	$\mathcal{CS}$ (1.)	$\mathcal{CS}$ (1.)
$\mathcal{RE}$	$\mathcal{RE}$ (2.)	$\mathcal{RE}$ (2.)	$\mathcal{RE}$ (2.)	$\mathcal{RE}$ (2.)

4 Arithmetic operations on languages

This section deals with the second main part of the paper: language properties of arithmetic operations on sets.

First we define some operations over sets of integers:

Definition 2 *Let $A, B \subseteq \mathcal{N}$ be two sets of integers, and let $c \geq 0$ be an integer. Let us define*

$$A+B=\{a+b \mid a\in A, b\in B\}, \qquad c+A=A+c=\{c\}+A,$$
$$A\cdot B=\{ab \mid a\in A, b\in B\}, \qquad c\cdot A=A\cdot c=\{c\}\cdot A,$$
$$A^B=\{a^b \mid a\in A, b\in B\}, \qquad A^c=A^{\{c\}}.$$

Theorem 2

	$c+A$	$c\cdot A$	$A+B$	$A\cdot B$	A^B
$\mathcal{REG}$	$\mathcal{REG}$	$\mathcal{REG}$	$\mathcal{REG}$	$\mathcal{CS}^*$	$\mathcal{CS}^*$
$\mathcal{CF}$	$\mathcal{CF}$	$\mathcal{CF}$	$\mathcal{CS}^*$	$\mathcal{CS}^*$	$\mathcal{CS}^*$
$\mathcal{CS}$	$\mathcal{CS}$	$\mathcal{CS}$	$\mathcal{CS}$	$\mathcal{CS}$	$\mathcal{CS}$
$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$

Let $a \geq 1$ be an integer. Then each inner cell of the table shows the Chomsky-class of the language $L_a(C)$, where $c \geq 0$, $A, B \subseteq \mathcal{N}$ such that $L_a(A)$ (and $L_a(B)$, if appropriate) belongs to the class shown in the heading of the row of the cell, and C is the result of the operation written in the heading of the column of the cell. With the exception of the elements marked with a $*$, the presented classes are the smallest ones in the Chomsky-hierarchy with this property.

Again, we prove this theorem by a series of lemmas. One of the key lemmas is the following:

Lemma 8 *Let $a \geq 1$ be an integer, and let $A, B \subseteq \mathcal{N}$ be two sets, such that the languages $L_a(A)$ and $L_a(B)$ are context-sensitive. Then the languages $L_a(A+B)$, $L_a(A\cdot B)$ and $L_a(A^B)$ are also context-sensitive.*

Proof sketch. We can use the same technique as the one used in the proof of Lemma 1. We start with two length-increasing grammar, that generate $L_a(A)$ and $L_a(B)$, respectively, and construct a grammar, that generates the resulting language.

For the language $L_a(A+B)$ we construct a grammar, that first generates words, which contain two encoded representation of words from $L_a(A)$ and $L_a(B)$, separated by some markers. Then the grammar decreases the value of the first word and increases the value of the second word one by one. Finally, the grammar erases the markers, and generate a terminal word. More formally, every terminal derivation fits to the following derivation scheme:

$$\begin{aligned} S &\Longrightarrow BS_1CMS_2E \Longrightarrow^* \\ &\Longrightarrow^* BX_{i_1} \dots X_{i_k} CMX_{j_1} \dots X_{j_l} E \Longrightarrow^* \\ &\Longrightarrow^* BX_{i'_1} \dots X_{i'_{k'}} CMX_{j'_1} \dots X_{j'_{l'}} E \Longrightarrow^* \\ &\Longrightarrow^* BCMX_{j''_1} \dots X_{j''_{l''}} E \Longrightarrow^* \\ &\Longrightarrow^* j''_1 \dots j''_{l''}, \end{aligned}$$

where

$$val_a(i_1 \dots i_k) + val_a(j_1 \dots j_l) = val_a(i'_1 \dots i'_{k'}) + val_a(j'_1 \dots j'_{l'}) = val_a(j''_1 \dots j''_{l''}).$$

The case of $L_a(A \cdot B)$ is a little bit more complex. Again, we construct a grammar, that first generates words, which contain two encoded representation of words from $L_a(A)$ and $L_a(B)$, separated by some marker. The multiplication can be done by repeated addition. We repeatedly decrease the value of the first word by one, and add the value of the second word to a third block of the generated word, which represents the result. When the value of the first word becomes zero, we erase the special markers, the second operand, the temporary storage and generate a terminal word. The operation of the grammar can be illustrated by the following derivation scheme:

$$\begin{aligned} S &\Longrightarrow BS_1CM_1S_2M_2X_0M_3X_0E \Longrightarrow^* \\ &\Longrightarrow^* BX_{i_1} \dots X_{i_k} CM_1X_{j_1} \dots X_{j_l} M_2X_0M_3X_0E \Longrightarrow^* \\ &\Longrightarrow^* BX_{i'_1} \dots X_{i'_{k'}} M_1X_{j'_1} \dots X_{j'_{l'}} DM_2X_{e_1} \dots X_{e_o} M_3BX_{f_1} \dots X_{f_p} E \Longrightarrow^* \\ &\Longrightarrow^* BCM_1X_{j_1} \dots X_{j_l} M_2X_0M_3BX_{f'_1} \dots X_{f'_{p'}} E \Longrightarrow^* \\ &\Longrightarrow^* f'_1 \dots f'_{p'}, \end{aligned}$$

where

$$\begin{aligned} val_a(i_1 \dots i_k) \cdot val_a(j_1 \dots j_l) &= val_a(i'_1 \dots i'_{k'}) \cdot val_a(j_1 \dots j_l) + \\ &+ val_a(j'_1 \dots j'_{l'}) + val_a(f_1 \dots f_p) = \\ &= val_a(f'_1 \dots f'_{p'}), \text{ and} \\ val_a(j_1 \dots j_l) &= val_a(j'_1 \dots j'_{l'}) + val_a(e_1 \dots e_0). \end{aligned}$$

The addition itself is more complex than it was in the grammar for the language $L_a(A+B)$, because we must somehow preserve the original operand. For that after each decrementation we increment the value of two blocks: one block will denote the final result, the other one is used to maintain information on the original operand. When the block of the operand becomes empty, we have the original value of it in another block. Then we do some tricks with the markers and zero out the temporary storage.

The case of $L_a(A^B)$ requires an additional step: we do the raising to the power by repeated multiplication, the multiplication by repeated addition and the addition by repeated incrementation. The solution uses 6 blocks. In the first block there is the first operand. In the second, there is the second operand in the beginning, but it gets decremented one by one during the derivation. In the remaining 4 blocks the grammar performs a multiplication, as described before. The details are rather long and needs no new techniques, therefore it is omitted here.

For all three of the constructed grammars, because the original grammars are length–increasing, the generated word is at least as long as any of the operands, and we always use a bounded number of special symbols, the constructed grammars have a workspace which size is at most a linear function of the length of the generated word, hence the generated languages are context–sensitive. □

The same constructions also work, when the representations of the operands are recursively enumerable. Now the grammars of the operands may not be length–increasing, and for that the constructed grammars may generate non–context–sensitive languages. This can be formulated as the next lemma:

Lemma 9 *Let $a \geq 1$ be an integer, and let $A, B \subseteq \mathcal{N}$ be two sets, such that the languages $L_a(A)$ and $L_a(B)$ are recursively enumerable. Then the languages $L_a(A+B)$, $L_a(A \cdot B)$ and $L_a(A^B)$ are also recursively enumerable.*

The following lemma states, that all of the Chomsky classes are closed under constant addition and multiplication.

Lemma 10 *Let $a \geq 1$ be an integer, and let $A \subseteq \mathcal{N}$ be a set, such that $L_a(A) \in \mathcal{F}$, where $\mathcal{F}$ is one of the classes $\mathcal{RE}$, $\mathcal{CS}$, $\mathcal{CF}$, $\mathcal{REG}$. Then for all $c \geq 0$ integer $L_a(c+A), L_a(c \cdot A) \in \mathcal{F}$.*

Proof sketch. The cases of $\mathcal{F} = \mathcal{CS}$ and $\mathcal{F} = \mathcal{RE}$ hold, because they are consequences of Lemmas 8 and 9, since $L_a(\{c\})$ is obviously regular. We can prove the remaining cases as follows:

It is known, that the classes $\mathcal{CF}$ and $\mathcal{REG}$ are closed under gsm–mappings and mirror images. We can easily construct two generalized sequential machines: G_1 and G_2, such that they depend on only a and c, and for all $u \in L_a(A)^{-1}$

$$val_a(G_1(u)^{-1}) = c + val_a(u^{-1}),$$

and

$$val_a(G_2(u)^{-1}) = c \cdot val_a(u^{-1}).$$

G_1 works exactly like humans do when add two numbers using paper and pencil. The constant c is coded in the internal structure of G_1, and the (mirror image) of the second operand is read from the input tape. The machine operates from right to left, digit by digit, handling the carry when necessary. The operation of G_2 differs a little from the way humans multiply: it also operates from right to left, but it multiplies one digit of the input by c and handles the carry. In this style the carry may be more than the base of the number system, but it is always less than c.

We get, that

$$L_a(c+A) = G_1(L_a(A)^{-1})^{-1},$$
$$L_a(c \cdot A) = G_2(L_a(A)^{-1})^{-1},$$

from which the theorem follows, because of the closure properties mentioned before. □

The following lemma deals with addition and multiplication of regularly representable sets:

Lemma 11 *Let $a \geq 1$ be an integer, and $A, B \subseteq \mathcal{N}$ be two sets, such that the languages $L_a(A)$ and $L_a(B)$ are regular. Then the language $L_a(A+B)$ is also regular. However, for all $a \geq 2$ bases there exist sets $A, B \subseteq \mathcal{N}$, such that the languages $L_a(A)$ and $L_a(B)$ are regular, but the language $L_a(A \cdot B)$ is nonregular.*

Proof. We prove the first part of the lemma first. The case of $a = 1$ follows from the fact, that the class of regular languages are closed under concatenation. To prove the case of $a \geq 2$ we construct a nondeterministic finite automaton, which accepts the language $(L_a(A+B))^{-1}$. From this, the case follows, because the class of regular languages are closed under mirror images.

Let $A_1 = (T, Q_1, \delta_1, s_1, F_1)$ be a deterministic finite automaton accepting $L_a(A)^{-1}$, and let $A_2 = (T, Q_2, \delta_2, s_2, F_2)$ be a DFA accepting $L_a(B)^{-1}$, where $T = \{0, 1, \ldots, n-1\}$. The constructed NFA will be the following (f and e are new symbols):

$$A_3 = (T, Q_3, \delta_3, (s_1, s_2, 0), F_3),$$

where

$$Q_3 = \{f\} \cup (Q_1 \cup \{e\}) \times (Q_2 \cup \{e\}) \times \{0,1\},$$
$$F_3 = \{f\} \cup (F_1 \cup \{e\}) \times (F_2 \cup \{e\}) \times \{0\},$$

and δ_3 is determined by the following formulas ($q_1, q_1' \in Q_1$, $q_2, q_2' \in Q_2$, $t \in T$, $c \in \{0,1\}$):

$$\begin{aligned}
\delta_3((q_1,q_2,c),t) \ni (q_1',q_2',c') \quad &\Leftrightarrow \quad \exists n,m \geq 0 : \\
&\delta_1(q_1,n) = q_1' \ \wedge \ \delta_2(q_2,m) = q_2' \ \wedge \\
&\wedge \ ((t = n+m+c \wedge c' = 0) \vee (a+t = n+m+c \wedge \ c' = 1)) \\
\delta_3((q_1,q_2,c),\lambda) \ni (\{e\},q_2,c) \quad &\Leftrightarrow \quad q_1 \in F_1
\end{aligned}$$

$$\begin{aligned}
&\delta_3((q_1,q_2,c),\lambda) \ni (q_1,\{e\},c) && \Leftrightarrow \quad q_2 \in F_2\\
&\delta_3((q_1,\{e\},c),t) \ni (q_1',\{e\},c') && \Leftrightarrow \quad \exists n \geq 0 \ : \delta_1(q_1,n) = q_1' \wedge\\
&\qquad \wedge\ ((t = n + c \wedge c' = 0) \vee (a + t = n + c \wedge c' = 1))\\
&\delta_3((\{e\},q_2,c),t) \ni (\{e\},q_2',c') && \Leftrightarrow \quad \exists m \geq 0 \ : \delta_2(q_2,m) = q_2' \wedge\\
&\qquad \wedge\ ((t = m + c \wedge c' = 0) \vee (a + t = m + c \wedge\ c' = 1))\\
&\delta_3((q_1,q_2,1),1) \ni f && \Leftrightarrow \quad q_1 \in F_1 \wedge\ q_2 \in F_2
\end{aligned}$$

The constructed automaton simulate both of the original ones. It tries to guess step by step the two operands of the addition, that result in the read number. Apart from state f, its states have three components: they contain the states of the original automata, and that whether there arose a carry bit in the last addition. When one of the enclosed automata stops, then the symbol e will be included in the state, instead of the internal state of the simulated automaton.

When the automaton is in a state (q_1, q_2, c), it has the following options:

- It reads a symbol t, and guess two numbers, n and m, such that the started addition can be continued by these numbers. It feeds the two simulated automaton with n and m, respectively. The new state will consist of the new states of the included automata, and the new carry bit.
- If one of the included automata is in a final state, it can decide, that the operand belonging to that automaton has ended. Afterwards, only the other internal automaton takes part in the addition.
- If there is a carry ($c = 1$), the read symbol is 1, and both of the included automata are in a final state, A_3 can decide, that both of the operands have ended. The new state will be f. This state has no following states. With this transition we can handle the case, when there is a carry in the final addition.

The automaton accepts a word, if after reading the whole number one of the following conditions is satisfied:

- the automaton is in state f, or
- both of the included automata are in a final state, and the carry bit is 0, or
- one of the included automata is in a final state, the other one has stopped before, and the carry bit is 0.

The details of the proof, that A_3 accepts exactly $(L_a(A + B))^{-1}$ are left to the reader.

To prove the second part of the lemma, let us consider the following example:

$$A := \left\{ \frac{a^k - 1}{a - 1} \;\middle|\; k \geq 1 \right\},$$
$$B := \{a^k + 1 | k \geq 1\}.$$

$L_a(A)$ and $L_a(B)$ can be expressed by the regular expressions 1^* and 10^*1, respectively, hence they are obviously regular. Let us suppose, that the language $L = L_a(A \cdot B)$ is regular. Then we can apply to L the pumping lemma concerning regular languages (Proposition 1). Let n be the constant appearing in the lemma. Let us consider the word $u = 1^n01^n$. (The word u is in L, because $1^n01^n = 1^n \cdot 10^n1$.) By the pumping lemma, we can iterate somewhere in the subword of the first n symbol of u, and for some $l \geq 1$, $u' = 1^{n+l}01^n \in L$. But this is a contradiction, because u' cannot be a base–a representation of a product of proper operands. □

By this lemma, we have finished the proof of Theorem 2. Like we did for Theorem 1, we repeat the table of Theorem 2, showing the lemmas, which belongs to the cells. The assertions that belongs to cells marked with a * are trivial consequences of lemma 8.

	$c+A$	$c \cdot A$	$A+B$	$A \cdot B$	A^B
$\mathcal{REG}$	$\mathcal{REG}$ (10.)	$\mathcal{REG}$ (10.)	$\mathcal{REG}$ (11.)	$\mathcal{CS}^*$	$\mathcal{CS}^*$
$\mathcal{CF}$	$\mathcal{CF}$ (10.)	$\mathcal{CF}$ (10.)	$\mathcal{CS}^*$	$\mathcal{CS}^*$	$\mathcal{CS}^*$
$\mathcal{CS}$	$\mathcal{CS}$ (10.)	$\mathcal{CS}$ (10.)	$\mathcal{CS}$ (8.)	$\mathcal{CS}$ (8.)	$\mathcal{CS}$ (8.)
$\mathcal{RE}$	$\mathcal{RE}$ (10.)	$\mathcal{RE}$ (10.)	$\mathcal{RE}$ (9.)	$\mathcal{RE}$ (9.)	$\mathcal{RE}$ (9.)

As a corollary of Theorem 2 we get results, which in some sense extend the Theorem of Horváth about the ranges of polynomials published in [13]:

Theorem 3 *Let $a \geq 1$ be a positive integer, $A_0, A_1 \subseteq \mathcal{N}$ finite sets, $X \subseteq \mathcal{N}$ a set for which $L_a(X) \in \mathcal{F}$, where $\mathcal{F}$ is one of the classes $\mathcal{RE}, \mathcal{CS}, \mathcal{CF}, \mathcal{REG}$. Then $L_a(A_1 \cdot X + A_0) \in \mathcal{F}$.*

Proof. It follows from the definitions, that

$$A_1 \cdot X + A_0 = \bigcup_{a_1 \in A_1, a_0 \in A_0} a_1 \cdot X + a_0.$$

By Theorem 10, we know that $a_1 \cdot X + a_0 \in \mathcal{F}$ for all $a_1, a_0 \geq 0$. From that the theorem follows, because all of the mentioned classes are closed under (finite) union. □

For higher ,,dimensions" we get weaker results, again as a direct consequence of Theorem 10.

Theorem 4 *Let $a \geq 1$ be a positive integer, $A_0, A_1, \ldots A_n \subseteq \mathcal{N}$ finite sets, $X \subseteq \mathcal{N}$ a set for which $L_a(X) \in \mathcal{F}$, where $\mathcal{F}$ is one of the classes $\mathcal{RE}, \mathcal{CS}$. Then $L_a(A_n X^n + \cdots + A_1 \cdot X + A_0) \in \mathcal{F}$.*

5 Conclusions

In this paper we have examined two aspects of the connection of formal languages and number theory. We got results for the problem of converting between canonical

number systems and performing arithmetic operations on sets, but there are many other open problems in this domain, we mentioned some in the text. We believe, that further research of this area can be fruitful for both fields.

Acknowledgement
The author wishes to thank László Hunyadvári for his useful suggestions.

References

[1] Alan Cobham. On the base-dependance of sets of numbers recognizable by finite automata. *Mathematical Systems Theory*, 3(2):186–192, 1969.

[2] Karel Culik II and Arto Salomaa. Ambiguity and decision problems concerning number systems. *Information and Control*, 56(3):139–153, 1983.

[3] Aldo de Luca and Antonio Restivo. Star-free sets of integers. *Theoretical Computer Science*, 43(2-3):265–275, 1986.

[4] Juha Honkala. Bases and ambiguity of number systems. *Theoretical Computer Science*, 31(1-2):61–71, May 1984.

[5] Juha Honkala. A decision method for the recognizability of sets defined by number systems. *RAIRO Inform. Thor. Appl.*, 20(4):395–403, 1986.

[6] Juha Honkala. A decision method for the unambiguity of sets defined by number systems. *The Journal of Universal Computer Science*, 1(9):648–653, September 1995.

[7] Christian Michaux and Roger Villemaire. Presburger arithmetic and recognizability of sets of natural numbers by automata: New proofs of Cobham's and Semenov's theorems. *Annals of Pure and Applied Logic*, 77(3):251–277, 19 February 1996.

[8] Arto Salomaa. *Formal Languages.* Academic Press, New York, 1973.

[9] J. Büchi. Weak second order logic and finite automata. *Z. Math. Logik, Grundlag. Math.*, 5:66–62, 1960.

[10] Grzegorz Rozenberg and Arto Salomaa. *Handbook of Formal Languages.* Springer Verlag, Berlin, Heidelberg, New York., 1997.

[11] Yehoshua Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonologie, Sprachwissenschaft und Kommunikationsforschung*, 14:113–124, 1961.

[12] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, N. Reading, MA, 1980.

[13] Horváth Sándor. Ranges of polynomials are non–context–free in any number system. To appear in: Pure Mathematics and Applications.

Acta Cybernetica **15** (2001) 137–149.

Parallel implementation for large and sparse eigenproblems*

E. M. Garzón† and I. García ‡

Abstract

This paper analyses and evaluates the computational aspects of an efficient parallel implementation for the eigenproblem. This parallel implementation allows to solve the eigenproblem of symmetric, sparse and very large matrices. Mathematically, the algorithm is supported by the Lanczos and Divide and Conquer methods. The Lanczos method transforms the eigenproblem of a symmetric matrix into an eigenproblem of a tridiagonal matrix which is easier to be solved. The Divide and Conquer method provides the solution for the eigenproblem of a large tridiagonal matrix by decomposing it in a set of smaller subproblems. The method has been implemented for a distributed memory multiprocessor system with the PVM parallel interface. A Cray T3E system with up to 32 nodes has been used to evaluate the performance of our parallel implementation. Due to the super-lineal speed-up values obtained for all the studied matrices, a detailed analysis of the experimental results is carried out. It will be shown that the management of the memory hierarchy plays an important role in the performance of the parallel implementation.

1 Introduction

Eigenproblems arise in a large number of disciplines of sciences and engineering. For example, they are used in: designing buildings, bridges and turbines; modeling queuing networks; analyzing stability of electrical networks; studying the fluid flow and so on. The matrices of these problems have a high dimension, a very low percentage of non-zero elements and, in general, they are non-symmetric. However, the symmetric eigenproblem constitutes the key in a lot of strategies to solve non-symmetric eigenproblems.

The computational cost and the memory requirements of the algorithms which provide a solution for the symmetric eigenproblem are very high when the matrix has a high dimension. In this context, the development of parallel algorithms and

*This work has been supported by the Ministry of Education of Spain (CICYT TIC99-0361).

†Department of Computer Architecture, University of Almería, 04120-Almería, Spain. e-mail: ester@ace.ual.es.

‡Department of Computer Architecture, University of Almería, 04120-Almería, Spain. e-mail: inma@ace.ual.es.

their efficient implementations on large scale supercomputer system are the only strategies which allow to solve this computationally expensive problem.

This paper deals with the parallel implementation of a strategy which provides a solution to the symmetric eigenproblem on a distributed memory multicomputer system. This strategy belongs to the so called direct methods and includes a divide and conquer technique.

The solution of the eigenproblem of a symmetric, large and sparse matrix $A \in R^{n \times n}$ can be obtained by the following transformations:

$$A = QTQ^T = QMDM^TQ^T = GDG^T \tag{1}$$

where D is a diagonal matrix whose non-zero elements represent the eigenvalues of A and T, and the columns of G and M are the eigenvectors of A and T, respectively.

The eigenproblem is solved by the following consecutive stages:

1. *Structuring the input matrix A.* This stage generates the tridiagonal matrix T and the orthonormal matrix Q, such that $A = QTQ^T$.

   The Lanczos Method with complete reorthogonalization is used at this stage. The complete reorthogonalization stage ensures the orthogonality of Q so that the spectrum of T is also the spectrum of A.

2. *Solving the eigenproblem of T.* At this stage the diagonal D and the orthogonal M matrices that give $T = MDM^T$ are the results of applying the Divide and Conquer method (DC). A divide-and-conquer method, developed by Cuppen [4], has been implemented at this stage. This method is decomposed in the following process:

   (a) Decomposing the eigenproblem of T in a set of $Sp_0 = 2^{Nv}$ subproblems $(T_1, T_2, \ldots, T_{Sp_0})$ of small dimension by rank-one transformations of T. Nv is the number of reconstruction levels.

   (b) Applying the QR method to every subproblem T_i. The corresponding eigenvalues and eigenvectors D_i and M_i, respectively, are obtained.

   (c) Reconstruction: From the set of matrices D_i and M_i $(i = 1, 2, \ldots, Sp_0)$, this procedure obtains the eigenvalue matrix D and the eigenvector matrix M. The reconstruction stage consists of a binary tree-structure of Nv levels.

3. *Computing the eigenvectors of A.* Let G be the orthonormal matrix whose columns g_i are the eigenvectors of A; i.e. $A = GDG^T$. This stage is solved by a matrix-matrix product, $G = QM$.

From the above algorithmic description, it can be seen that the parallel implementation of the eigenproblem consists of three consecutive parallel stages. Each stage manages a different kind of data structure. The first stage (Lanczos method) deals with an irregular sparse matrix; the second stage involves computations on a set of tridiagonal sub-matrices (structured data) and a set of dense matrices; the third stage is simply a product of dense matrices.

The parallel implementation of the first stage (Lanczos) is based on a decomposition in domains of the input sparse matrix which includes a computationally inexpensive pre-processing stage, namely Pivoting-Block [9]. This preprocessing stage guarantees that the input data partitions and its associated computations are balanced.

The parallel implementation of the Divide and Conquer method is based on a decomposition in domains of the input and output data [13]. The binary tree of tasks is distributed among Processing Element (PE) so that the same number of branches of the tree is allocated to each PE. When the number of PEs of the multiprocessor system P verifies that $P < Sp_0$, each PE starts the reconstruction process independently until the number of sub-problems is equal to P. When $P \geq Sp_0$, a set of PEs collaborates for the solution of a pair of subproblems. As the reconstruction stages evolve, the dimension of the subproblems are greater and the number of PEs collaborating for the same pair of subproblems increases.

The parallel product of matrices ($G = QM$) is carried out starting with a partition of Q and M by rows among PEs. We have implemented a strategy which reduces the memory requirements.

The main contributions of this paper consists of: (a) providing a parallel implementation for large sparse eigenproblems by linking the above described stages; (b) evaluating the proposed parallel implementation using a wide variety of problems and (c) doing a computational analysis to determine which factors are responsible for the super-lineal speed up values obtained from our experimental results.

This paper is organized as follows. In Section 2, the mathematical foundations of the applied method are briefly introduced. In Section 3, parallel implementations of every stage is described. Finally, in Section 4, new and non-standard performance indicators for evaluating parallel implementations are defined. Moreover, experimental results of the performance evaluation of our parallel implementation are shown and discussed. Performance evaluations were carried out by a multicomputer (Cray T3E) using no more than 32 processing elements.

2 Describing the Applied Methods

The eigenproblem of large and sparse matrices is solved by a direct method which allows to determine the matrix decompositions described by (1).

The Lanczos method, briefly described in Subsection 2.1, is applied to obtain T and Q (tridiagonal and orthogonal matrices, respectively). The eigenproblem of T is solved by the Divide and Conquer method [11] (Subsection 2.2), which obtains matrices D and M from T, where D is a diagonal matrix whose elements are the eigenvalues of T and A, and columns of $G = QM$ are the eigenvectors of A.

2.1 Structuring Sparse Matrix

The Lanczos Method with Complete Reorthogonalization has been used for finding a structured matrix with the same spectrum as the input sparse matrix. The

Lanczos method is considered an effective method for obtaining, from a symmetric matrix A, a symmetric tridiagonal matrix T and a set of orthonormal vectors, q_j $(0 < j \leq n)$. Given a symmetric matrix $A \in R^{n \times n}$ and a vector q_1 with unit norm, the Lanczos method generates the orthonormal matrix Q and the tridiagonal matrix T, in n iterative steps. This method is discussed in [1, 3, 12, 14]. The outer loop of the Lanczos algorithm is an iterative procedure (index j) which at the *j-th* iterative step computes the α_j and β_j coefficients of the tridiagonal matrix T and the vector q_{j+1}, where α_j and β_j denote the elements of the main and secondary diagonals of T, respectively. This loop includes a sparse matrix-vector product and a reorthogonalization process. The reorthogonalization procedure used in this work is the so called complete reorthogonalization (CR). CR is computationally expensive but it allows to ensure that the eigenvalues of A and T are the same. The Lanczos algorithm is particularly appropriate for structuring sparse matrices of high dimension.

2.2 Solving the eigenproblem of T

A solution for the eigenproblem of a tridiagonal matrix based on the Divide and Conquer method (DC) was proposed by Golub [11] and, lately, developed by Bunch, Nielsen and Sorensen [2] and Cuppen [4].

The key of this method consists of dividing the input matrix of high dimension into several sub-matrices of lower dimension and solving the eigenproblem of high dimension from the solutions of eigenproblems of lower dimension. This strategy is very useful to solve the eigenproblem of very high dimensions. The DC method can be described as in Algorithm 1.

Algorithm 1 Divide and Conquer Algorithm: $DC(T) \rightarrow D, M$

1 *do* $i = 1, \ldots, Sp_0;\ i + 1$
2 Div $(i) \rightarrow T_i$ # SubDivision #
3 QR$(T_i) \rightarrow D_i, M_i$ # Solving Small Eigenproblems #
4 $Sp = Sp_0$;
5 *do* $k = 1, \ldots, Nv;\ k + 1$ # Reconstruction Levels #
6 *do* $i = 1, \ldots, Sp - 1;\ i + 2$ # Reconstructing Couples of SubMatrices #
7 Reconstruction $\rightarrow \hat{D}_{i/2}, \hat{M}_{i/2}$
8 First Deflation
9 Second Deflation
10 Solve Secular Equation $\rightarrow \hat{D}_{i/2}, V$
11 $\hat{M}_{i/2} = \begin{pmatrix} M_i & 0 \\ 0 & M_{i+1} \end{pmatrix} V$ # Intermediate matrix-matrix Product#
12 $Sp = Sp/2$

The subdivision process by rank one modifications of T is associated with line 2. This process generates the set of sub-matrices (T_i). Then, the eigenproblem of

every T_i is solved by the QR method [12] which generates the D_i (eigenvalues) and M_i (eigenvectors) matrices. (Sp_0 denotes the number of initial subproblems).

The loop with index k is associated with the reconstruction process (lines 5-12), where k denotes the level of reconstruction. The total number of levels of reconstruction is given by $Nv = log_2(Sp_0)$. The level of reconstruction k includes $Sp/2$ reconstruction processes for a couple of sub-matrices (T_i, T_{i+1}) whose eigenvalues and eigenvectors are known (D_i, M_i and D_{i+1}, M_{i+1}). The outputs of this process are $\hat{D}_{i/2}, \hat{M}_{i/2}$. These matrices are the solution of the eigenproblem for the sub-matrix which is the result of the association of a couple of tridiagonal submatrices (T_i, T_{i+1}). Details about the reconstruction process are described in [4]. This process may include deflations which reduce the computational cost of the secular equation solution and the dimensions of the matrices which are included in the so called Intermediate matrix-matrix Product (line 11). The number of subproblems at level k is denoted by Sp. When $Sp = 2$ the eigenproblem of T is solved ($\hat{D}_{i/2} = D, \hat{M}_{i/2} = M$).

2.3 Computing the eigenvectors of A

The matrix-matrix product $G = QM$ is computed in order to obtain the eigenvectors of the input matrix A. This last stage, which completes the solution of the eigenproblem of A, is computationally very expensive and needs large memory requirements. However, if the goal were only to compute the spectrum of the input matrix, this stage could be omitted.

3 Parallel Implementation

The method for solving the eigenproblem of a symmetric sparse matrix, discussed in Section 2, has an extremely high computational cost. Furthermore, this method demands large memory requirements. Consequently, its implementation on a distributed memory multiprocessor is necessary, specially, when the input matrix A is of high dimension.

The parallel implementation of the method has been carried out using a SPMD programming model and the PVM standard library. The whole solution of the eigenproblem (LDC) has been divided into a set of procedures: *Lanczos*, *DC* and *Final Product matrix-matrix*. These procedures link their executions in a sequential way because the data dependences prevent several procedures from simultaneous execution. Thus, every procedure must be independently parallelized.

3.1 The Lanczos Method

In the solution of the eigenproblem, the *Lanczos* method is the only procedure which manages irregular data structures; i.e. a sparse input matrix A. Since the Lanczos algorithm works on mixed computations (dense-sparse), special care must be taken in the data distribution among processors in order to optimize the

work load balance for computations on both dense and sparse data structures. A data distribution called *Pivoting Block* [9] is used to balance the computational load of this procedure. Pivoting Block estimates a permutation of the rows of A obtaining an homogeneous density of the non-zero elements. Thus, the classical block partition applied to the permuted matrix is able to obtain a similar number of non-zero elements for the sparse sub-matrix allocated at every PE. Consequently, computations linked to dense and sparse structures are balanced. Details about parallel implementation of *Lanczos* algorithm can be found in [7, 8]. The outputs of this parallel algorithm are: the tridiagonal matrix T and the orthonormal matrix Q. The tridiagonal matrix T is stored at the local memory of every Processing Element (PE). When the parallel *Lanczos* procedure finishes, every PE stores $\lceil \frac{n}{P} \rceil$ rows of Q at its local memory, where P is the number of PEs in the multiprocessor system.

3.2 The *DC* Method

An efficient parallel implementation of *DC* method on share memory multiprocessors has been proposed by Dongarra and Sorensen [5]. Moreover, parallel implementations on a distributed memory multicomputer have been described by Ipsen and Jessup [13] and Tisseur and Dongarra [15]. In our implementation we have used most of the ideas described in [13].

The structure of the *DC* algorithm suggests a natural way to split and distribute the computational work among the set of PEs. The *DC* method can be represented by a binary tree of tasks which can be decomposed into P sets of tasks and distributed among PEs.

As an illustration, the example in Figure 1 starts with T subdivided into $SP_0 = 16$ sub-matrices and a multiprocessor system with $P = 4$ is considered. As it can be seen in Figure 1, for a problem which is subdivided into 16 subproblems, the *DC* method consists of 4 reconstruction levels. The reconstructions at levels $k = 1$ and 2 are carried out by every isolated PE. When $k = 3$, two groups of two PEs are defined. Thus, every group of PEs cooperates in the reconstruction of a couple of sub-matrices. At the final level ($k = 4$), the four PEs cooperate in the last reconstruction.

At the end of this stage, the non-zero elements of D and the rows of M are distributed among the set of PEs.

3.3 The Final matrix-matrix product

In order to complete the solution of the eigenproblem of the input matrix A, the matrix G is computed by the matrix-matrix product $G = QM$. The parallel implementation of this matrix-matrix product is more difficult than the standard one because every processor allocates only a subset of rows of Q and M. However, the communication time and the memory requirements have been optimized by re-using data structures defined and used at previous stages.

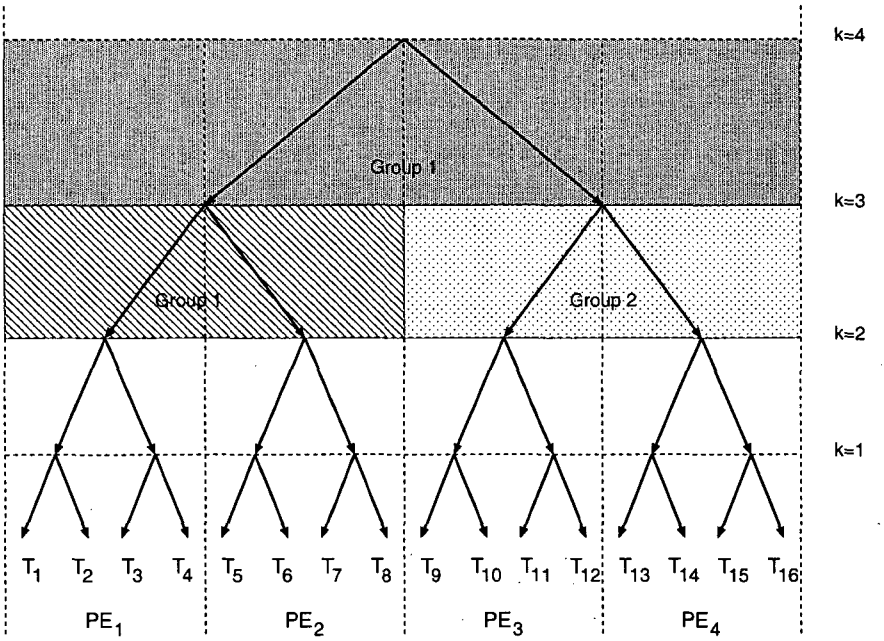


Figure 1: Binary tree of tasks for the *DC* method, and task distribution among PEs for a multiprocessor system with $P = 4$.

4 Evaluation

The evaluation of the parallel implementation for the solution of the eigenproblem has been carried out on a multiprocessor system Cray T3E using a set of n dimensional input matrices. Some of the matrices belong to the Harwell-Boeing collection of test matrices [6]. Moreover, a subset of the test matrices has been designed to analyze the parallel implementation of the *DC* method. These test matrices are obtained by permutating a set of tridiagonal matrices denoted by [1,μ,1] or [1,k,1] [13], where $\alpha_k = k\mu$ or $\alpha_k = k$, respectively, and $\beta_k = 1$ $(k = 1 \ldots n)$. These matrices are denoted by "tn", where n is the dimension of the input matrix. The selection of the initial Lanczos vector (q_1) allows us to control the number of deflations the *DC* method produces. Consequently, it is possible to generate test problems with a high or a low computational cost for the *DC* method.

In Table 1, three matrices of the set of test matrices used in the evaluation of our parallel implementation are characterized by parameters like the dimension of the matrix (n) and the percentage of non-zero elements (γ). The last two columns of Table 1 provide numerical results which specify the accuracy of the applied methods. Specifically, numerical results for the parameters $log_{10}\frac{R}{\|A\|_F}$ and $log_{10}Ort$ are given; where R is the norm of the residual related to the eigenproblem solution $(R = \| AG - GD \|_F)$ and $Ort = \| G^TG - I \|_F$ provides a measurement of the

orthogonality of the eigenvectors ($\| \cdot \|_F$ denotes the Frobenius matrix norm).

Table 1: Numerical results of the accuracy of LDC method for several test matrices.

Matrix	n	γ	$log_{10} \frac{R}{\|A\|_F}$	$log_{10} Ort$
BFW782B	782	0.9 %	-15	-12
t1024	1024	0.3 %	-14	-13
t2048	2048	0.1 %	-14	-12

The parallel performance evaluation was carried out executing the algorithm with $P = 1, 2, 4, 8, 16$ and 32 PEs and subdividing the tridiagonal matrix T into P sub-matrices; i.e. $Sp_0 = P$ for the DC method. However, executions using only one or a few PEs were only possible for some of the test matrices (the smaller ones) because of the fact that the computer ran out of memory for large matrices; for example for the matrix $t7168$ the multiprocessor system ran out of memory when less than 16 PEs were used, so execution times were only obtained for $P = 16$ and 32 PEs. Under these circumstances it was not possible to compute the standard values of the speed-up for evaluating the performance of the parallel implementation. As an alternative to the speed-up, we have defined a new parallel performance estimator called Incremental Speed-up ($IncSpUp$) which provides information about how much the computing time diminishes when the number of PEs increases. $IncSpUp$ is defined as follows:

$$IncSpUp(2^i) = \frac{T(P = 2^{i-1})}{T(P = 2^i)}, \tag{2}$$

where T(P) is the run time of the execution with P PEs. For ideal parallel implementations, the value of the Incremental Speed-up should be equal to $IncSpUp = 2$, which corresponds to a lineal speed-up [10].

The experimental values for the Incremental Speed-up obtained from executions of our parallel implementation have been plotted in Figure 2. A set of eight matrices whose dimension n ranges between 782 and 7168 was used as test matrices; two of the matrices belong to the Harwell-Boeing collection (BFW728B and BCSSTK27), the remaining test matrices belong to the above described kind of matrices (tn). For every tn matrix the parallel algorithm was run twice; one of them producing many deflations and the other few deflations. As it was previously described, many or few deflations may appear depending on the value of the initial Lanczos vector (q_1). In Figure 2, tn and tn^* graphs correspond to the same matrix but for execution with few and many deflations, respectively.

From Figure 2 performance of the parallel implementation can be analyzed for every value of the number of PEs, P. For every tested matrix such that $n \geq 2048$, the values of the $IncSpUp$ estimator were greater than 2 when $2 \leq P \leq 16$ but for $P = 32$ only the largest matrices ($t5120$ and $t7168$) gave $IncSpUp \geq 2$. From the definition of the $IncSpUp$ it is easy to see that from the values of the $IncSpUp$ for $2, 4, \ldots, P$ PEs, it is possible to obtain the value of the Speed-up for P PEs because

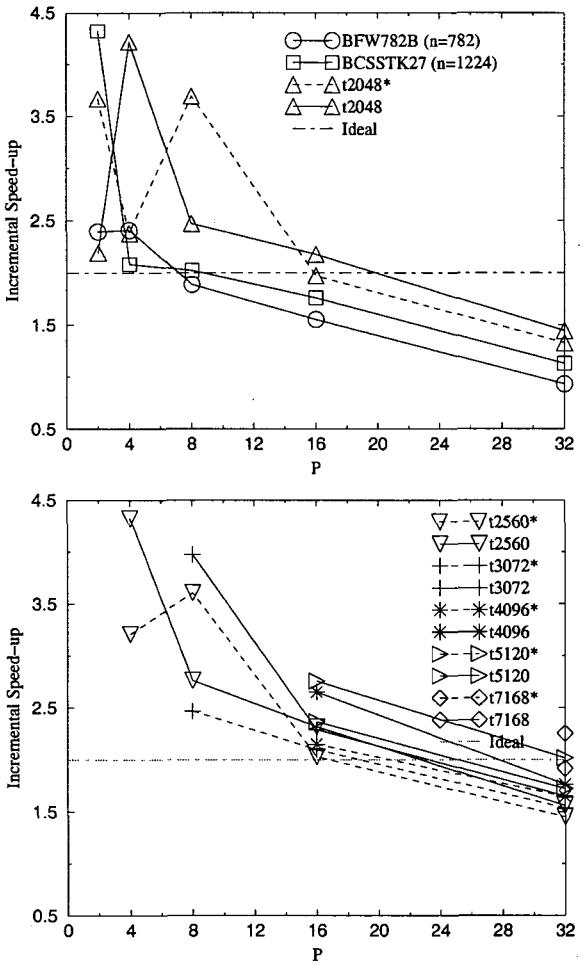


Figure 2: Incremental Speed-up of the parallel execution of LDC against the number of processors (P) for several input matrices.

$SpUp(P) = IncSpUp(P) \times IncSpUp(P/2) \times \ldots \times IncSpUp(2)$. The values of $IncSpUp$ estimator obtained in our experimental results are equivalent to efficiency values higher than 1, it means that our implementation exhibits a super-lineal behavior.

Notice that, as the number of PEs increases, the computational work load of every processor diminishes but the interprocessor communications and delays for synchronizing tasks do not decrease but even may increase. Values of the $IncSpUp$ less than 2 can have been produced as a consequence of long delays for synchronization, which are mainly due to work load unbalances among processors, or long

interprocessor communications.

In an attempt to determine the causes for both the super-lineal Speed-up behavior as well as the decreasing of the $IncSpUp$ when the number of PEs increases, a detailed analysis of the performance was carried out. This analysis was made through a pair of additional parameters: the number of cache faults and the execution profiles. Execution profiles provide measurements of the percentage of the computational work related to every procedure involved in the parallel algorithm. Experimental results will show that the management of the memory hierarchy plays an important role in algorithms with large memory requirements.

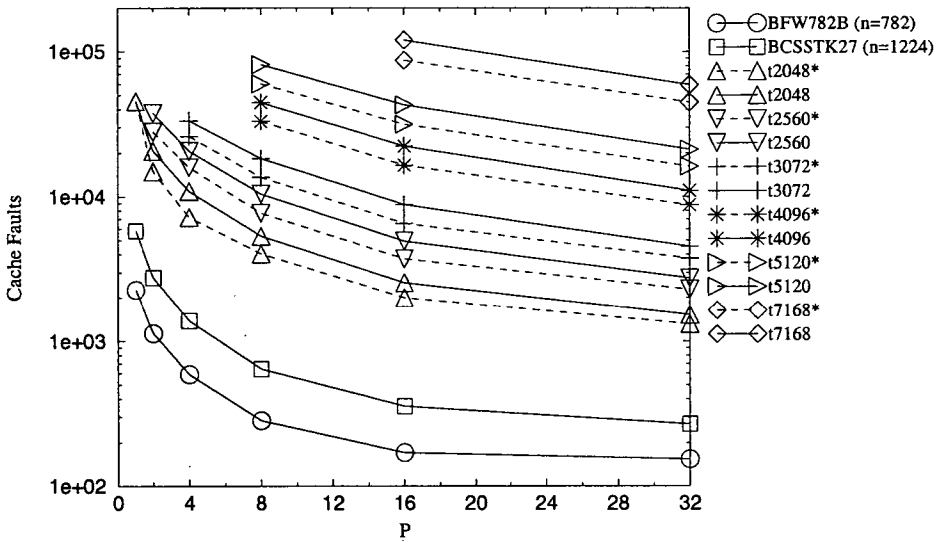


Figure 3: Cache faults versus the number of PEs for several dimensions of input matrices. (*) means many deflations.

Figure 3 shows a log plot of the number of cache faults against P. It can be seen that the number of cache faults diminishes considerably as P increases and this fact is more relevant for small values of P. This is mainly due to the fact that the percentage of the total data that can be allocated at the cache memory is greater when more PEs are used. This means that the time spent on accessing to data memory decreases as the number of PEs increases. This justifies that super-lineal Speed-up values ($IncSpUp \geq 2$) have been obtained in our experimental results.

Experimental results for execution profiles are shown in the Table 2. The procedures included in the LCD algorithm that have stronger computational work load are:. *Lanczos* (lc), QR method (QR), intermediate product of matrices (IP, line 11 of DC) and final product of matrices (FP). The characters in brackets are referred to the notation of the column head in Table 2. Moreover, this table has two additional columns which specify the percentages of the run time related to the waiting time for synchronization (wt) and the interprocessor communication process (c).

Table 2: Execution profile of *LDC* for several test matrices. *lc, FP, IP, QR, wt* and *c* denote the percentage of the work load associated to the Lanczos method, the Final Product of matrices, the Intermediate Product of matrices, the QR method, the wait for messages time and the interprocessor communications, respectively.

	Many Deflations					*Few Deflations*					
P	*lc*	*FP*	*QR*	*wt*	*c*	*lc*	*FP*	*IP*	*QR*	*wt*	*c*
	t2048										
2	39	55	2	3	-	24	33	25	2	14	-
4	30	66	-	1	-	32	25	33	-	6	-
8	46	45	-	4	1	33	32	21	-	9	-
16	42	42	-	2	10	33	33	18	-	4	6
32	28	27	-	5	31	23	23	11	-	6	26
	t3072										
2	47	48	-	3	-	32	32	25	1	7	-
4	55	42	-	1	-	23	27	30	-	17	-
8	46	47	-	3	-	32	32	23	-	9	-
16	45	44	-	1	6	34	33	19	-	4	4
32	33	33	-	4	14	27	27	14	-	5	18
	t5120										
8	49	47	-	1	-	28	26	36	-	6	-
16	47	46	-	-	3	33	31	21	-	9	2
32	38	38	-	3	15	30	30	17	-	5	11
	t7168										
16	50	45	-	-	2	30	29	28	-	9	-
32	42	41	-	2	10	32	31	18	-	6	8

On the left side of Table 2, the results are associated with executions of the *DC* procedure that include many deflations, so the reconstruction process and the intermediate products (*IP*) are not very hard from a computational point of view. On the right hand side, we can see the results associated with the *DC* procedure that includes few deflations. So, the intermediate products represent a relatively large percentage of the total computational work.

From data in Table 2, it can be seen that the communication processes are computationally irrelevant except for execution with $P = 16$ and $P = 32$, but their importance decreases when n increases.

Notice that the values of the *wt* parameter are also estimations of the work load balance among processors since a synchronization stage always precedes every communication among processors. For all the analyzed cases the value of *wt* is extremely small. Thus, the parallel implementation has a good work load balance. In [13], from the point of view of parallel implementation, deflations have been described as a serious drawback, as they can produce load unbalance. Nevertheless, the values of *wt* obtained in our experimental results show that deflations do not produce a relevant work load unbalance.

5 Conclusions

In this paper a parallel implementation of the eigenproblem of symmetric sparse and large matrices is proposed and evaluated. The solution is based on a direct method which mainly consists of three consecutive stages. Parallel implementations of every isolated stage have been described in the bibliography [7, 8, 13, 15]. However, a parallel implementation for the whole eigenproblem solution which includes these methods has not been reported anywhere. Our proposal is able to provide a solution for very large matrices which can not be solved with a uniprocessor system due to both the high computational complexity and the large memory requirements. We have solved all the problems associated to the work load unbalance which frequently appear when sparse matrices are involved in parallel computations.

A detailed analysis of the parallel implementation has been carried out through the experimental values of the Incremental Speed-up, the number of cache faults and the execution profiles. It has been proved that the designed parallel implementation is very efficient since it includes specific devices which allow: (a) distributing the computational work load associated with all the procedures in a balanced way; (b) establishing interprocessor communications that do not increase considerably the run time, and what is more, (c) improving the mamory data access time, especially for irregular data.

References

[1] Berry, M.W. Large-scale sparse singular value computations. *The International Journal of Supercomputer Applications*, 6(1):13–49, Spring 1992.

[2] Bunch, J.R.; Nielsen C.P. and Sorensen, D.C. Rank one modification of the symmetric eigenproblem. *Numerische Mathematik*, 31(1):31–48, 1978.

[3] Cullum, J.K. and Willoughby, R.A. *Lanczos algorithms for large symmetric eigenvalue computations*, volume 1: Theory, volume 2: Programs. Birkhäuser, Stuttgart, 1985.

[4] Cuppen, J.J.M. A divide and conquer method for the symmetric eigenproblem. *Numerische Mathematik*, 36:177–195, 1981.

[5] Dongarra, J.J. and Sorensen, D.C. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM Journal on Scientific and Statistical Computing*, 8(2):139–154, March 1987.

[6] Duff, I.S.; Grimes, R.G. and Lewis, J.G. User's guide for the Harwell-Boeing sparse matrix collection. Technical report, Research and Technology Division, Boeing Computer Services, 1992.

[7] García, I; Garzón, E.M.; Cabaleiro, J.C.; Carazo, J.M. and Zapata, E.L. Parallel tridiagonalization of symmetric matrices based on Lanczos method. In

M. Valero, E. Onate, M. Jane, J. L. Larriba, and B. Suarez, editors, *Parallel Computing and Transputer Applications*, pages 236–245, Amsterdam, 1992. IOS Press.

[8] Garzón, E.M. and García, I. Parallel implementation of the Lanczos method for sparse matrices: Analysis of data distributions. In ACM, editor, *FCRC '96: Conference proceedings of the 1996 International Conference on Supercomputing*, pages 294–300, New York, 1996. ACM Press.

[9] Garzón, E.M. and García, I. Evaluation of the work load balance in irregular problems using Value Based Data Distributions. *Proceedings of the IASTED International Conference Parallel and Distributed Systems. Euro-PDS'97*, pages 137–143, 1997.

[10] Garzón, E.M. and García, I. A parallel implementation of the eigenproblem for large, symmetric and sparse matrices. In J.J. Dongarra, E. Luque, and T. Margalef, editors, *Recent advances in PVM and MPI*, volume 1697 of *Lecture Notes in Computer Science*, pages 380–387. Springer-Verlag, 1999.

[11] Golub, G.H. Some modified matrix eigenvalue problems. *SIAM Review*, 15(2):318–344, April 1973.

[12] Golub, G.H. and Van Loan C. F. *Matrix computations.* Johns Hopkins Studies in the Mathematical Sciences. The Johns Hopkins University Press, Baltimore, MD, USA, third edition, 1996.

[13] Ipsen, I.C.F. and Jessup, E.R. Solving the symmetric tridiagonal eigenvalue problem on the hypercube. *SIAM Journal on Scientific and Statistical Computing*, 11(2):203–229, March 1990.

[14] Simon, H.D. Analysis of the symmetric Lanczos algorithm with reorthogonalization methods. *Linear Algebra and its Applications*, 61:101–131, 1984.

[15] Tisseur F. and Dongarra J. A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures. *SIAM Journal on Scientific Computing*, 20(6):2223–2236, November 1999.

Acta Cybernetica 15 (2001) 151–162.

Parallelization of an algorithm for the automatic detection of deformable objects*

J.M. González-Linares†, N. Guil†, E.L. Zapata†
P.M. Ortigosa‡, and I. García‡

Abstract

This work presents the parallelization of an algorithm for the detection of deformable objects in digital images. The parallelization has been implemented with the message passing paradigm, using a master-slave model. Two versions have been developed, with synchronous and asynchronous communications.

1 Introduction

Numerous real life applications of image analysis need to detect the presence and localization of certain objects, which suffer deformations due to several factors, like sampling errors or the own flexibility of the material. Diverse methods that allow to recover these objects exist, like free-form models based on "snakes" [1] or the parametric models [2], [3]. Free-form models have the disadvantage that they cannot be used reliably in an automatic environment, and parametric models need an initial segmentation of the images.

In this work an algorithm has been selected that combines the Generalized Hough Transform (GHT, [4], [5]) with a deformation model [6], to form an objective function that is minimized by a stochastic global optimizer. The GHT provides an automatic mechanism and is highly immune to occlusions, noise and cluttering, thus solving the problems associated with other methods. The whole method can be represented by the Bayesian rule, where the prior information is formed by a template of the object to be detected and a group of deformations that are applied on the template. The likelihood is obtained from the likeness measure that provides the GHT, while the *a posteriori* information is given by the application of the Bayes rule. The inference on this bayesian model is carried out by a maximum *a*

*This work has been supported by the Ministry of Education of Spain (CICYT TIC99-0361).

†Departamento de Arquitectura de Computadores, Campus de Teatinos, Universidad de Málaga, E-29080 Málaga (Spain). E-mail: gonzalez,nico,ezapata@ac.uma.es.

‡Departamento de Arquitectura de Computadores y Electrónica, Universidad de Almería, E-04120 Almería (Spain). E-mail: pilar,inma@iron.ualm.es.

posteriori estimator (MAP), therefore an optimization algorithm named UEGO (Universal Global Evolutionary Optimizer, [7]) have been selected. This optimization algorithm allows to calculate the global maximum in an efficient way.

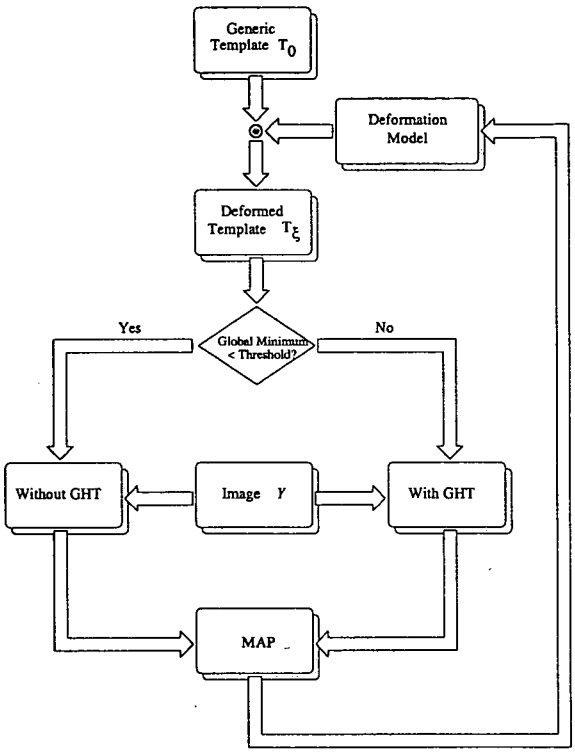


Figure 1: Outline of the method for the automatic detection of deformable objects

The computational complexity of this method is very high, since the evaluation of the GHT requires a lot of computation. The GHT allows to calculate the parameters that define the similarity Euclidean transformations (rotation, scale and displacement), but these parameters are not necessary to be computed when a good enough solution, below a given threshold, is found. To reduce the computational requirements two objective functions are used; the first one evaluates the GHT while the second not, so its computational complexity is much smaller. This second objective function can be applied when the global minimum found is below a threshold. The outline of operation of the method can be seen in Figure 1. Even with this simplification, the computational complexity remains quite high, therefore its parallelization is very interesting to obtain reasonable computational times.

Figure 2 shows one of the test images used in the evaluation of the algorithm. The top-left image presents a complex scene where the guitar is to be detected.

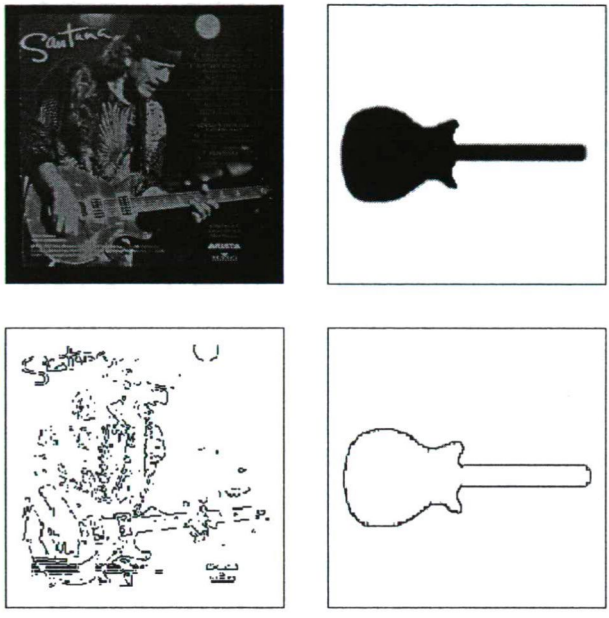


Figure 2: *Test image and contour edges*

The top-right image corresponds to the template used in the detection and the two bottom images are the contour edges obtained with the Canny detector [8]. The deformation model is applied over the template contour to obtain a deformed template T_ξ. Then, the GHT obtains the rotation, scale and displacement parameters of T_ξ in the image. After applying these parameters over the deformed template, a likeness measure can be computed:

$$\mathcal{E} = \frac{1}{n_T} \sum (1 + \Phi(x,y)|\cos(\beta(x,y))|) \tag{1}$$

where

$$\Phi(x,y) = -\exp\left(-\rho\sqrt{(\delta_x^2 + \delta_y^2)}\right), \tag{2}$$

and n_T is the number of edge points in the template, δ_x, δ_y is the distance between the contour points of the deformed template and image, $\beta(x,y)$ is the difference between their gradient angles, and ρ is a parameter to control the smoothness of the function. The value of $\mathcal{E}$ is normalized between 0 and 1, with 0 corresponding to a perfect match.

The energy value provided by (1) is used as an objective function that is minimized by UEGO. Experimental results show that near a minimum the similarity transformation parameters remain equal, thus the GHT can be skipped to speed up the process. Several intermediate results along with the final result obtained are presented in Figure 3. In the bottom-right image the final result can be seen,

Figure 3: *Example of the results obtained with the algorithm*

where the contour of the deformed, rotated, scaled and displaced template has been superimposed on the original image.

2 UEGO

UEGO is a stochastic optimization algorithm that looks for groups of solutions to optimize them by means of a particular method.

In the used implementation SASS (Single Agent Stochastic Search [9]) has been selected as local optimizer. This method presents the advantage that the objective function is not required to be continuous or differentiable. The operation consists in generating a sequence of random paths. These paths start from a search point that suffer a perturbation with a random variable. This random variable can have different probability distributions, being the Gaussian distribution the most efficient and the one that has been implemented. SASS is quite efficient to find local minima, but it presents the disadvantage of being trapped in local minima. The introduction of SASS inside an evolutionary algorithm as UEGO allows solving this problem.

The basic concept in UEGO is the one of *species* that is defined as a region (window) in the search space. Its *center* and its *radius* determine this region, where the center is a solution or point in the search space, and the radius is a positive number that indicates the width of the region. This definition assumes that the

concept *distance* exists inside the search space. UEGO algorithm creates a list of species, and each species is optimized by means of a certain method (SASS).

The role of this window is to 'localize' the optimizer that is called by a species and can 'see' only its window, so every new sample is taken from here. This means that the largest step made by the optimizer in a given species is not larger than the radius of the given species. If the value of a new solution is better than that of the old center, the new solution becomes the center and the window is moved.

The radius of a species is not arbitrary; it is taken from a list of decreasing radii, the *radius list*. The radii decrease in a regular fashion in geometrical progression. The first element of this list is always the diameter of the search space (r_1), which will ensure that the largest species always contains the whole space independently of its center. The diameter is given by the largest distance between any two possible solutions according to the distance mentioned above, and it is an input parameter. If the radius of a species is the ith element of the list, then we say that the *level* of the species is i.

During the optimization process, a list of species is kept by UEGO. The algorithm is in fact a method for managing this *species-list* (i.e. creating, deleting and optimizing species).

The creation of the list of species consists in, initially, to calculate a random point inside the search space. This point will be the center of a new species whose radius comes determined by the diameter of the search space r_1. The list of species is finite and limited by the input parameter M that indicate the maximum number of species. This initial list of species is optimized by SASS.

In the species list not only the species optimization is carried out, but also three additional types of processes ([10]). The first process is a mutation process (species creation mechanism), where a species can be divided in two or more species if more than one local minimum exists inside the species.

The second process type is one of fusion. If the centers of two species are separated by a smaller distance than the radius associated with the actual level i, the two species combine into a single one. The center of the new species will correspond to the best (minimal) center of both species, and its radius will be the biggest one.

The third process carried out is elimination. It has already been indicated that the maximum number of species is limited by a parameter. If the number of species, as a consequence of the previous processes, overcomes this parameter, the species that have been created more recently are eliminated.

All these processes, together with the optimization of the species, are carried out in an iterative way like the one indicated in Figure 4. The number of iterations l (levels) is another input parameter. Two further input parameters exist: the allowed maximum number of objective function evaluations (N) and a threshold ν that controls the maximum distance a species can travel in the level i. Although five input parameters exist, only four of them are necessary, since the fifth parameter will be obtained by means of the application of some principles. In [10], all these principles are described in detail.

```
Initialize species
Optimize species
For i=2...1,
    Create species
    Fuse species
    Eliminate species
    Optimize species
    Fuse species
```

Figure 4: *The* UEGO *code*

3 Parallelization of the method

The high computational requirements of the optimization algorithms have raised the appearance of numerous parallelization strategies. One of the most common strategies consists of carrying out global parallelization following a *master-slave* model. The *master* processor takes charge of making global decisions and distributing the populations of points that must be evaluated. On the other hand the *slave* processors evaluate the objective function in those populations of points. Another common strategy is the *coarse grain* parallelization, where the different processors execute the same algorithm of optimization on different subpopulations in an independent way. Though the executions are independent, intermediate results are sometimes exchanged. In the *fine grain* strategy the evaluations of the objective function are distributed among the processors. This strategy is quite common when working with massively parallel machines.

In this work, two parallelization strategies of UEGO based on a *master-slave* model have been evaluated. The first one (PSUEGO, [11]) presents several synchronism points so that the *slaves* processors send the evaluation of their lists of species to the *master* processor. The second version (PAUEGO) eliminates the synchronism points to avoid waiting times.

3.1 Parallel synchronous implementation (PSUEGO)

The *slaves* processors only need to receive the two own features of any species (i.e. its center and its radius), from the *master* processor, to be able to run the *optimize* and *create_species* procedures, so the amount of information involved in the communication procedures is quite small.

The *optimize* and *create_species* procedures do not need any additional information; each procedure only depends on a species and does not depend on another parameters or species. For this reason, these procedures can be run independently in several *slaves* processors at the same time.

In the initialization phase, the *master* processor forms the initial species list containing not a single point, but NP random points, where NP is the number

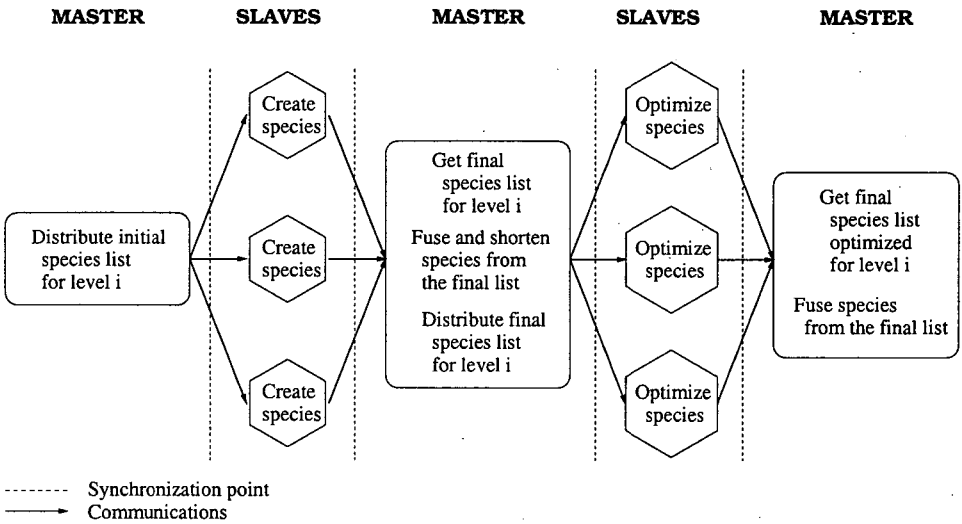


Figure 5: Synchronous version of UEGO

of *slave* processors. These NP points will be the centers of the first NP species. Then it distributes the species among the *slave* processors so all species can be optimized simultaneously. When they finish, they send the results to the *master* processor so that it forms the list of species that will be used in the following level. Then an iterative process (l levels) is carried out whose operation outline for this synchronous version can be observed in Figure 5. The oval boxes represent instructions that are executed in the *master* processor and the hexagonal boxes in the *slave* processors. The vertical dotted lines indicate synchronism points and the continuous lines indicate communications where the arrow shows the direction of the communication.

Initially, the *master* processor distributes the list of species among the *slave* processors. The *slave* processors pick up the species and evaluate them, trying to create new species. Meanwhile, the *master* processor stays in a wait state until **all** the *slave* processors finish creating species. Once the *master* processor has received all the new species, it applies the fusion and elimination processes to them in order to complete the final species list at this level. Later, it distributes this list among the *slave* processors, which take charge of optimizing each of their assigned species. When all species have been optimized, they are sent to the *master* processor that applies a fusion process and forms the species list that will be used in the following iteration.

When evaluating the objective function, the algorithm decides what objective function type must be used depending on the value of the reached minimum. If the *slave* processors act independently, there will be processors that evaluate more often the objective function with GHT than others, which will evaluate the objective

function without GHT a bigger amount of times. To avoid this computational unbalance, the *slave* processors send the rotation, scale and displacement parameters to the *master* processor, computed for their best minimum, every time they communicate with the *master* processor. The *master* processor will select the best minimum so far and, if it is below the threshold, it will communicate to all *slave* processors the parameters that should evaluate the objective function without GHT. Using this technique all the *slave* processors approximately evaluate the same number of times the objective functions with and without GHT.

Anyway, the distribution of the computational load is not well balanced. The evaluation of the objective function is carried out in the species creation and optimization processes, which are only executed in the *slave* processors. Therefore, the *master* processor is most of the time waiting results from the *slave* processors. Due to the fact that the performance of this synchronous version is very low, the elimination of some synchronism points is necessary. In this way the *master* processor can also work in the creation and optimization processes and the processors can distribute the computational load in a more dynamic way.

3.2 Parallel asynchronous implementation (PAUEGO)

The second parallel strategy (PAUEGO) is intended to solve some of the previous problems. In this new implementation the load has been balanced forcing the *master* processor to optimize and create species while the *slave* processors are working. Another change consists of the reduction of several points of synchronization. Now the *master* processor can start to carry out the synchronous operations over the species before it has received all the information, so the idle time can be reduced considerably.

The first modification that has been carried out with regard to the synchronous version is located in the initialization phase. Now every *slave* processor has a species initialization procedure; in particular, every *slave* processor chooses two or more points as centers of new species and later on, it optimizes them. Once the species is optimized, the *slave* processor creates and sends a new sublist of species for each of the optimized species to the *master* processor. On the other hand, the *master* processor only initializes a single species (not NP species like in the case of PSUEGO), and later it optimizes the species and creates new species from the optimized one. Once this new sublist has been created, the *master* processor is prepared to receive any information (sublists of species) from any *slave* processor. If at some time the *master* processor does not receive any sublist, then it begins to fuse the lists of species. This fusion process stops when the *master* processor receives any sublist from any *slave* processor, and goes on when no more reception of information from any *slave* processor is produced at that time.

Once the *master* processor has received all the sublists created by the *slave* processors and the *fuse* procedure has been applied, it starts the iterative loop. Figure 6 shows how the work is distributed among the processors. The meaning of the used symbols is the same as in Figure 5.

The iterative part of PAUEGO for the *slave* processors does not have any modifi-

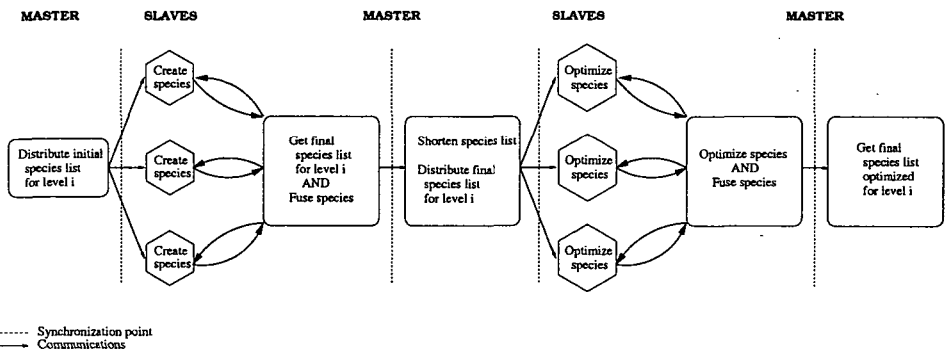


Figure 6: Asynchronous version of UEGO

cation with respect to the iterative part of PSUEGO, but there are some changes for the *master* processor. In this version, the *master* processor always checks the arrival of information (a new created sublist of species or an optimized species) from the *slave* processors and if any information has arrived it sends a new species. Otherwise the *master* processor contributes to the optimization process in such a way that when the algorithm is in the species creation stage, the *master* processor fuses the received species, and when the algorithm is in the optimization phase, then the *master* processor optimizes a species. These species must be species that have not been sent to any *slave* processor in this optimization phase. These processes executed by the *master* processor are always interrupted when new information arrives to the *master* processor, and later they are resumed.

As in the previous version, all the processors communicate the rotation, scale and displacement parameters, obtained for the best minimum, so that all processors do a similar number of evaluations of the GHT.

4 Results

The two versions of the algorithm have been tested on a SGI/CRAY T3E machine, with 32 DEC 21164 (Alpha EV-5) processors at 300 MHz and 128 MB of RAM memory each. The implementation has been carried out using the PVM message passing library [12]. To obtain the speedup, both versions have been executed 20 times on a group of five different images, being carried out a total of 100 executions. Each of these tests has been made for 2, 4, 8 and 16 processors. In addition, the sequential algorithm has been executed on the same machine to be able to compare it with the results obtained by the parallel versions. The DEC 21164 processors have two cache levels. The first level has a cache of 8 KB for the data and another of 8 KB for the instructions, and the second level has a cache of 96 KB for data and instructions. This relatively small size of the cache penalizes the performance of the algorithm in comparison with a processor like the Pentium II whose second

level cache has a size of 512 KB. The spatial locality of the data takes advantage of the cache of 512 KB. For this reason, the sequential algorithm executed on the Pentium II can be up to three or four times quicker than on the Alpha 21164.

The selected group of images presents different characteristics, in the sense that there are images where the objective function with GHT is applied few times and in others the opposite happens. This is because for some images a good optimum better than the threshold is reached very quickly and in others it is not possible to reach it. Due to these differences, the computation times are very disparate, varying from few seconds to several minutes. Nevertheless, the obtained speedup remains approximately constant independent of the computational complexity.

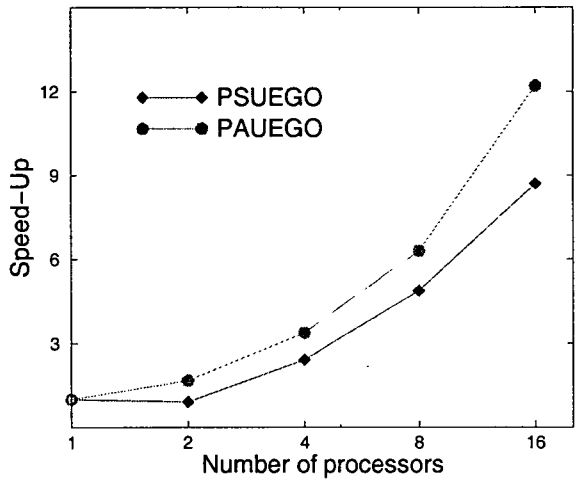


Figure 7: *Speedup for* PSUEGO *and* PAUEGO

In Figure 7 the average results obtained for both parallelizations are presented. The results with one processor correspond to the sequential version. From these results the following considerations are made:

- The distribution of the computational load is made at the species level, then it is convenient that the number of species is a multiple of the number of processors to obtain a good balance. Although the concrete number of species depends on the input data (the image and the template), the parameter M can be used to limit the number of species.

- The objective functions for this concrete application present numerous local optima [13], therefore the number of evaluations of the objective function for each species has a high variation. Typically, the total number of function evaluations is between 2000 and 3000. Then, although the distribution of species is well balanced, the computational load for each processor is not necessarily such.

- The selection of the objective function type that is evaluated depends on the best approximate global optimum. If several species are evaluated in parallel, it is possible to find global optima that are better than the given threshold before the sequential version. For this reason the average number of evaluations of the objective function with GHT in the parallel versions can be smaller to that of the sequential version.

The results obtained for PSUEGO show a low speedup because the *master* processor does not collaborate in the evaluation of the objective function. In addition, the distribution of the load is not well balanced, so the processors are waiting to each other most of the time (30%−40% of the total time). With PAUEGO the results improve because of the dynamic distribution of the load and because the *master* processor also works in the species creation and optimization processes. Although the distribution of the species is well balanced, the processors spend a 10% − 20% of the total time in waits and communications. The reason for this unbalance is that the evaluation of different species can have quite different computational complexities.

5 Conclusions

The parallelization of an algorithm for the automatic detection of deformable objects has been presented. A *master-slave* model has been selected and two versions, a synchronous one and an asynchronous one, have been implemented. The asynchronous version obtains a better speedup thanks to the dynamic distribution of the load and the participation of the *master* processor in the tasks with more computational complexity. The obtained speedup does not come closer to the ideal one due to the granularity of the problem. A distribution of the computational load with a finer grain would imply that the granularity is at the level of function evaluations instead of species. It would allow a better computational balance, but the number of communications would be much higher and the speedup would not improve.

References

[1] M. Kass, A. Witkin, and D. Terzopoulos, "Snakes: Active contour models", *International Journal of Computer Vision*, 1(4), 1988, 321–331.

[2] L.H. Staib and J.S. Duncan, "Boundary finding with parametrically deformable models", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(11), 1992, 1061–1075.

[3] A.L. Yuille, P.W. Hallinan, and D.S. Cohen, "Feature extraction from faces using deformable templates", *International Journal of Computer Vision*, 8(2), 1992, 133–144.

[4] D.H. Ballard, "Generalizing the Hough Transform to Detect Arbitrary Shapes", *Pattern Recognition*, 14(2), 1981, 111–122.

[5] N. Guil, J.M. González-Linares, and E.L. Zapata, "Bidimensional shape detection using an invariant approach", *Pattern Recognition*, 32(6), 1999, 1025–1038.

[6] Y. Amit, U. Grenander, and M. Piccioni, "Structural image restoration through deformable template", *Journal of the American Statistical Association*, 86(414), 1991, 376–387.

[7] M. Jelasity, P.M. Ortigosa, and I. García, "UEGO, an Abstract Clustering Technique for Multimodal Global Optimization", Accepted for publication in the *Journal of Heuristics*.

[8] John Canny, "A computational approach to edge detection", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6), 1986, 679–698.

[9] D.C. Karnopp, "Random search techniques for optimization problems", *Automatica*, 1, 1963, 111–121.

[10] Pilar M. Ortigosa, *Stochastic Global Optimization Methods. Parallel Processing.*, PhD Thesis, University of Málaga, 1999.

[11] P.M. Ortigosa, I. García, and M. Jelasity, "Two Approaches for Parallelizing the UEGO Algorithm", Accepted for publication in *Mátraháza Optimization Days*, Kluwer Academic Publishers, 2001.

[12] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM 3 Users guide and reference manual*, Technical Report, Oak Ridge National Laboratory, 1993.

[13] J.M. González-Linares, N. Guil, E.L. Zapata, P.M. Ortigosa, and I. García, "Deformable Shapes Detection by Stochastic Optimization", in Proceedings of the *IEEE Int'l Conf. on Image Processing (ICIP'2000)*, Vancouver, Canada, September 10–13, 2000.

Acta Cybernetica **15** (2001) 163-172.

An online scheduling algorithm for a two-layer multiprocessor architecture *

Cs. Imreh†

Abstract

In this paper we give online algorithms and competitive ratio bounds for a scheduling problem on the following two-layer architecture. The architecture consists of two sets of processors; within each set the processors are identical while both the processors themselves and their numbers may differ between the sets. The scheduler has to make an online assigment of jobs to one of the two processor sets. Jobs, assigned to a processor set, are then sceduled in an optimal offline preemptive way within the processor set considered. The scheduler's task is to minimize the maximum of the two makespans of the processor sets.

1 Introduction

In the most fundamental parallel machine scheduling model, we have a sequence of jobs, each of them has a processing time, and we have to process them on the available uniform machines. A schedule specifies for each job a machine and a time interval on the machine when the job is processed on it. The length of the time interval must be the processing time, the starting and ending points of the time interval are called the *starting and finishing time* of the job. A schedule is *feasible* if for each machine the time intervals do not overlap. Our goal usually is to minimize the maximal finishing time. Sometimes it is allowed to preempt the jobs. In this case, we have to specify for each job a sequence of machines with not overlapping time intervals (one machine can have more time intervals), and the total length of the time intervals must be the processing time. For more details on scheduling problems we refer to [6].

The most fundamental example of an online machine scheduling problem is the online problem of jobs arriving one by one. In this problem we have a fixed number m of identical machines. The jobs and their processing times are revealed to the online algorithm one by one. When a job is revealed, the online algorithm must irrevocably assign the job to a machine, without any information about the further

*This work has been supported by the Grant OTKA T030074

†Department of Informatics, University of Szeged, Árpád tér 2, H-6720 Szeged, Hungary e-mail: cimreh@inf.u-szeged.hu

jobs. The starting time of the job is the finishing time of the previous job assigned to the machine considered. By the load of a machine, we mean the sum of the processing times of all jobs assigned to the machine. Then, the maximal completion time is the maximum load, thus the objective is to minimize the maximum load, often called the *makespan*. The first result for this online scheduling model is due to Graham [4], the best algorithm which is known for this problem can be found in [1], but the best possible competitive ratio is still unknown for $m \geq 3$. For details and results on other online scheduling problems we refer to the survey [8].

In this paper we consider a scheduling problem where the machines form a two-layer multiprocessor architecture. In this problem we have two sets $\mathcal{P}$ and $\mathcal{S}$ of identical machines containing k and m machines with $k \leq m$. The jobs arrive one by one. Each job j has two different processing times p_j and s_j, one for each set of machines. We assign the jobs online to one of the two machine sets. Finally, when the stream of jobs has come to an end, we schedule the jobs assigned to $\mathcal{P}$ (respectively, the jobs assigned to $\mathcal{S}$) on the machines of $\mathcal{P}$ (respectively, $\mathcal{S}$) in an offline way so as to minimize the preemptive makespan. Let C_P (respectively, C_S) denote this optimal makespan. The cost of the schedule, which we want to minimize, is the maximum of the makespans, $\max\{C_P, C_S\}$. We denote this problem by $LS(k,m)$ (*Layered Scheduling*). The general problem without fixing the numbers k and m is denoted by LS. It is worth noting that the $LS(1,1)$ problem is the online two-machine scheduling problem with unrelated machines which problem is investigated in [2].

A similar problem called classification with preemptive scheduling (or CPS for short) is studied in [5]. Just as in the LS problem, in CPS an online scheduling to one of two sets of identical processors is followed by an offline preemptive scheduling within the processor sets. However in CPS the objective to be minimized is the sum of the two makespans instead of their maximum. For the CPS problem two algorithms are developed the first one is a greedy algorithm. The competitive ratio of this greedy algorithm is linear in m/k, therefore this algorithm can be effective only in the cases when m/k is small. A more difficult algorithm with constant competitive ratio is also presented.

In this paper we study the algorithms which are presented in [5], and we determine their competitive ratios for the LS problem. Moreover, we prove a general lower bound, namely, we show that no online algorithm can have smaller competitive ratio than 1.781.

The paper is organized as follows. In the following section we introduce some basic notation for the LS problem. Then, in Section 3, we present two algorithms for solving the problem and determine the competitive ratios of these algorithms. In Section 4 we present two lower bounds for LS. First, we construct input sequences for each fixed pair of values (k,m) to show that no online algorithm for the fixed (k,m) can be better than $(1+\sqrt{5})/2$-competitive. Then we construct a single input sequence to show that for all online algorithms there exists a pair (k,m) such that the algorithm cannot be better than $(3+\sqrt{17})/4$-competitive. Finally, we summarize the results and present some open questions regarding the problem in Section 5.

2 Notation and preliminaries

In the problem considered we have two sets of machines, $\mathcal{P}$ is the set of k identical machines and $\mathcal{S}$ is the set of m identical machines. In what follows we assume that $k \leq m$. Furthermore, each job j has two processing times, p_j if we schedule it on the machines of $\mathcal{P}$ and s_j if we schedule it on the machines of $\mathcal{S}$. Here we allow ∞ processing time, which means that it is not possible to process the job on the machines in the set. The vector (p_j, s_j) is called the *size* of the job. The algorithm has to decide in an online fashion on the arrival of each job where to assign it. When the sequence of jobs is finished, we schedule the jobs assigned to $\mathcal{P}$ respectively $\mathcal{S}$ on the machines of $\mathcal{P}$ respectively $\mathcal{S}$, in the way that we minimize the makespan preemptively in an offline way.

With preemption a job may be scheduled on multiple machines. In preemptive scheduling we have to assign time intervals to each job on one or more machines. The total length of the intervals must be the processing time of the job, and if time intervals $(q_1, t_1), \ldots, (q_i, t_i)$ are assigned to a job, then $t_j \leq q_{j+1}$ must be valid for $j = 1, \ldots, i-1$. Furthermore, no two jobs may have overlapping intervals on the same machine. It is well-known and one can easily see that for any set of jobs, the preemptive makespan of the optimal scheduling is the maximum of the maximal processing time and the load of the jobs. By the *load of a set of jobs* we mean the value obtained by dividing the sum of the processing times by the number of machines.

For any subset I of the jobs, we use the following notation

$$S_I = \sum_{j \in I} s_j, \quad P_I = \sum_{j \in I} p_j, \quad \text{Smax}_I = \max_{j \in I} s_j, \quad \text{Pmax}_I = \max_{j \in I} p_j.$$

Using these notation, the cost of a schedule SC can be written in the form

$$w(SC) = \max\{\frac{P_R}{k}, \text{Pmax}_R, \frac{S_Q}{m}, \text{Smax}_Q\},$$

where R and Q are the sets of jobs assigned to $\mathcal{P}$ and $\mathcal{S}$, respectively.

The optimal cost on a list L of jobs is denoted by $OPT(L)$. This is the minimum of the costs of the schedules which assign the jobs of L to the sets. We measure the performance of the presented algorithms by the competitive analysis. For this reason let A be an arbitrary online algorithm and let $A(L)$ denote the cost of the schedule produced by A on a list L of jobs. An algorithm A is called *c-competitive* if for every list L of jobs $A(L)$ is at most c times greater than OPT(L). The *competitive ratio* of an algorithm on a problem is c if the algorithm is c-competitive and it is not $\bar{c}$-competitive for any $\bar{c} < c$.

3 Upper bounds

There is a simple online algorithm for LS. The basic idea is to assign each job to the set of machines where the job has a smaller load. This algorithm is called load

greedy, (in short LG) and it is presented in [5] for the problem CPS. It has a similar analysis in our case. Formally, the algorithm is given as follows:

Algorithm *LG*: *When a job j arrives, then it is assigned to $\mathcal{P}$ if $\frac{p_j}{k} \leq \frac{s_j}{m}$, otherwise, it is assigned to $\mathcal{S}$.*

We can prove the following statement.

Theorem 1 *Algorithm LG has the competitive ratio* $\max\{2, m/k\}$ *on problem* $LS(k,m)$.

Proof. Consider an arbitrary list L of jobs, and denote by a and b the makespans obtained by LG on $\mathcal{P}$ and $\mathcal{S}$, respectively. Suppose that $a \geq b$. If the makespan is defined on $\mathcal{P}$ by the maximal completion time, then denote the job with this maximal processing time by j. Then $p_j = a$, and since our algorithm assigns this job to $\mathcal{P}$, we get that, $s_j > \frac{m}{k}a$. Hence, the optimal cost is at least a, and this yields that we have an optimal solution. Now, suppose that the makespan is defined by the load of the jobs. Let P denote the set of the jobs assigned by LG to $\mathcal{P}$, and let R and Q denote the sets of the jobs from P which are assigned to $\mathcal{P}$ and $\mathcal{S}$ in an optimal solution, respectively. Then, the optimal cost is at least $\max\{\frac{P_R}{k}, \frac{S_Q}{m}\}$. Furthermore, by the definition of LG, we get that $\frac{P_Q}{k} \leq \frac{S_Q}{m}$. This yields that the cost of the optimal solution is at least $\max\{\frac{P_R}{k}, \frac{P_Q}{k}\} \geq a/2$, and our statement follows.

Let us assume that $a < b$. Now, consider two cases depending on the makespan on $\mathcal{S}$. If the makespan is the load, then in the same way as above, we obtain that the optimal cost is at least $b/2$, which yields that the algorithm is 2-competitive. Suppose that the makespan is defined by a maximal processing time. Denote the job with this maximal processing time by j. Then, $s_j = b$ and since our algorithm assigns this job to $\mathcal{S}$, we obtain that $p_j > \frac{k}{m}b$. Hence, the optimal cost is at least $\frac{k}{m}b$, and our statement follows.

To prove that the above analysis is tight, consider one job of size $(1, \frac{m}{k} - \varepsilon)$. Algorithm *LG* assigns this job to $\mathcal{S}$ with cost $\frac{m}{k} - \varepsilon$, hence, since the optimal cost is 1, by choosing a sufficiently small ε, the competitive ratio on this job is arbitrarily close to $\frac{m}{k}$. To prove that the bound 2 is tight, we have to consider a sequence of jobs where the load of each job is the same in the two sets.

□

Algorithm LG works well only in the cases when m/k is small. Here we study an algorithm which is also efficient when m/k is large. This algorithm is defined in [5] and can be considered as a generalization of the reject total penalty type algorithms which are presented in [3] and [7]. The algorithm has two parameters $0 < \alpha \leq 1$ and $0 < \gamma \leq 1$.

Algorithm $A(\alpha, \gamma)$

- 1. *Initialization.* Let $R := \emptyset$.
- 2. When job j arrives:
  - (i) If $\frac{p_j}{k} \leq \frac{\gamma}{m} \cdot s_j$, then assign j to $\mathcal{P}$.
  - (ii) Let r be the cost of the optimal offline preemptive scheduling of the set $R \cup \{j\}$ on $\mathcal{P}$. Formally, $r = \max\{\frac{P_{R\cup\{j\}}}{k}, \text{Pmax}_{R\cup\{j\}}\}$. If $r \leq \alpha \cdot s_j$, then
    - (a) Assign j to $\mathcal{P}$,
    - (b) Set $R = R \cup \{j\}$.
  - (iii) Otherwise, assign j to $\mathcal{S}$.

Theorem 2 *The competitive ratio of algorithm $A(\alpha, \gamma)$ is c on problem LS, where*

$$c = \max\{1 + \frac{1}{\alpha}, 1 + \alpha + \gamma, 1 + \frac{1}{\gamma}\}.$$

Proof. We prove this statement in two parts. First, we show that the algorithm is c-competitive, and later we prove that it is not better than c-competitive. Let us consider an arbitrary sequence of jobs, and denote the list of the jobs by L. Fix an optimal schedule of the jobs. Denote by P_{opt} the set of jobs assigned to $\mathcal{P}$ in the optimal schedule. Let P_0 be the set of jobs with $\frac{p_j}{k} \leq \frac{\gamma}{m} \cdot s_j$, and P be the set of jobs assigned to $\mathcal{P}$ by our algorithm. Let us observe that, by the definition of the algorithm, we have $P_0 \subseteq P$. Define the following sets

$$X = L \setminus (P_{opt} \cup P), \quad Y = P_{opt} \setminus P,$$

$$Z = P_{opt} \cap (P \setminus P_0), \quad U = P_{opt} \cap P_0,$$

$$V = P_0 \setminus P_{opt}, \quad W = (P \setminus P_0) \setminus P_{opt}.$$

Then, the algorithm gives the following cost on L:

$$A(L) = \max\{\frac{P_Z + P_U + P_V + P_W}{k}, \text{Pmax}_Z, \text{Pmax}_U,$$

$$\text{Pmax}_V, \text{Pmax}_W, \frac{S_X + S_Y}{m}, \text{Smax}_X, \text{Smax}_Y\}.$$

Furthermore, the optimal cost is

$$OPT(L) = \max\{\frac{P_Y + P_Z + P_U}{k}, \text{Pmax}_Y \text{Pmax}_Z,$$

$$\mathrm{Pmax}_U, \frac{S_X + S_V + S_W}{m}, \mathrm{Smax}_X, \mathrm{Smax}_V, \mathrm{Smax}_W\}.$$

To prove the first part of the theorem, we have to show that $A(L) \leq c \cdot OPT(L)$. In the proof we will use the following lemma which is proved in [5]. For the reader's convenience, we recall here this result and also sketch the proof of the statement.

Lemma 3 *([5]) The following inequalities are valid:*

$$(1) \quad \frac{P_W}{k} \leq \alpha \cdot \mathrm{Smax}_W,$$

$$(2) \quad \mathrm{Pmax}_W \leq \alpha \cdot \mathrm{Smax}_W,$$

$$(3) \quad \alpha \cdot \mathrm{Smax}_Y \leq \max\{\frac{P_Z + P_W + P_Y}{k}, \mathrm{Pmax}_{Z \cup W \cup Y}\}.$$

Proof. We first prove (1). Let j be the last job from W. At the time when it was assigned to $\mathcal{P}$ by the algorithm, we had $r \leq \alpha \cdot s_j$. On the other hand, j is the last job in W, thus $\frac{P_W}{k} \leq r$. Furthermore, obviously $s_j \leq \mathrm{Smax}_W$, and the validity of (1) follows. We can prove (2) in the same way as (1). Indeed, let j be a job from W with $p_j = \mathrm{Pmax}_W$. When it was assigned to $\mathcal{P}$, we had $p_j \leq r \leq \alpha \cdot s_j$. This inequality yields (2).

There exists a job $j \in Y$ with $s_j = \mathrm{Smax}_Y$. At the time when it was assigned to $\mathcal{S}$, we had $r > \alpha s_j$. On the other hand, $R \cup \{j\} \subseteq Z \cup W \cup Y$ was valid for the considered set R, thus $r \leq \max\{\frac{P_Z + P_W + P_Y}{k}, \mathrm{Pmax}_{Z \cup W \cup Y}\}$ was also valid by definition. Therefore, the required inequality holds.

□

Using this lemma, we can prove the desired upper bound. For this reason consider the following cases.

Case 1: Suppose that $A(L) = \max\{\mathrm{Pmax}_Z, \mathrm{Pmax}_U, \mathrm{Smax}_X\}$. In this case $A(L) \leq OPT(L)$, and thus, the algorithm results in an optimal solution.

Case 2: Suppose that $A(L) = \frac{P_Z + P_U + P_V + P_W}{k}$. In this case the definition of the set V yields that $\frac{P_V}{k} \leq \frac{\gamma}{m} \cdot S_V \leq \gamma OPT(L)$. On the other hand $\frac{P_Z + P_U}{k} \leq OPT(L)$. Furthermore, by Lemma 3, we have that $P_W/k \leq \alpha\ \mathrm{Smax}_W \leq \alpha OPT(L)$. Therefore, $A(L) \leq (1 + \alpha + \gamma)OPT(L)$.

Case 3: Suppose that $A(L) = \mathrm{Pmax}_V$. Then, by the definition of the set V, $\mathrm{Pmax}_V/k \leq \gamma\ \mathrm{Smax}_V/m$. Since $k \leq m$, this yields that $\mathrm{Pmax}_V \leq \gamma\ \mathrm{Smax}_V \leq \gamma OPT(L)$. This is possible only when $\gamma = 1$, and in this case the algorithm results in an optimal solution.

Case 4: Suppose $A(L) = \mathrm{Pmax}_W$. Then by Lemma 3, $\mathrm{Pmax}_W \leq \alpha\ \mathrm{Smax}_W \leq \alpha \cdot OPT(L)$, therefore, this case is possible only if $\alpha = 1$, and the algorithm gives an optimal solution.

Case 5: Suppose that $A(L) = \frac{S_X+S_Y}{m}$. In this case, by the definition of the set Y we have that $\frac{S_Y}{m} \leq \frac{P_Y}{\gamma k} \leq OPT(L)/\gamma$. Hence, we obtain that $A(L) \leq (1+1/\gamma)OPT(L)$.

Case 6: Suppose that $A(L) = \text{Smax}_Y$. By Lemma 3 this yields that

$$A(L) \leq \frac{1}{\alpha}\max\{\frac{P_Z+P_W+P_Y}{k}, \text{Pmax}_{Z\cup W\cup Y}\}.$$

Consider now three subcases. If $A(L) \leq \frac{P_Z+P_W+P_Y}{\alpha k}$, then since $\frac{P_W}{\alpha k} \leq \text{Smax}_W \leq OPT(L)$ (by Lemma 3), we get that $A(L) \leq (1+1/\alpha)OPT(L)$. If $A(L) \leq 1/\alpha\ \text{Pmax}_{Z\cup Y}$, then we obtain immediately that $A(L) \leq OPT(L)/\alpha$. Finally, if $A(L) \leq 1/\alpha\ \text{Pmax}_W$, then by Case 4, we have that $A(L) \leq OPT(L)$.

Since we considered all the possible cases, we proved that the algorithm is c-competitive. We can prove that the bound c is tight by the following examples.

First, assume that $k = 1$ and $m > \gamma \cdot (1+\alpha)/\alpha$. Consider the following sequence of two jobs. The first job has size $(\alpha \cdot M, M)$, the second job has size $(M+\varepsilon, M(1+\alpha)/\alpha)$ for some large M and small ε. By the definition of m, we have that $\alpha M > \gamma M/m$, and thus, the first job is assigned to $\mathcal{P}$ in Step (ii). The second job is assigned to $\mathcal{S}$. Therefore, the cost of the algorithm is $M(1+\alpha)/\alpha$ on this sequence. The optimal cost is $M+\varepsilon$, we assign the first job to $\mathcal{S}$ and the second to $\mathcal{P}$. As M tends to ∞ the ratio of these costs tends to $1+1/\alpha$, hence we proved that the first bound is tight.

To prove the tightness of the second bound, fix the value of k and let m be much greater than k. Consider the following sequence of jobs. First consider $M(m-k)$ jobs of size $(\gamma \cdot k/m, 1)$. The second part of the sequence contains k jobs of size (M, ∞), finally, the third part contains k jobs of size $(\alpha \cdot M, M)$. Then the first and second parts are assigned to $\mathcal{P}$ in Step (i), the third part is also assigned to $\mathcal{P}$ in Step (i) or in Step (ii). Therefore, the cost of the algorithm is

$$\frac{M(m-k)\gamma k/m + Mk + \alpha Mk}{k}.$$

The optimal solution assigns the first and the third parts to $\mathcal{S}$, and the second part to $\mathcal{P}$ and its cost is M. As m tends to ∞, the ratio of these costs tends to $1+\alpha+\gamma$, hence, we proved that the second bound is tight.

To prove that the third bound is tight, consider such k and m that satisfy the inequality $\alpha/k > \gamma/m$ and the following sequence of jobs. The first part of the sequence is one job of size $(\alpha(m/(\gamma k)+2\varepsilon), (m/(\gamma k)+2\varepsilon))$. The second part contains Mk jobs of size $(1, m/(\gamma k)+\varepsilon)$, and the third part contains m jobs of size (∞, M). Then the algorithm assigns the first job to $\mathcal{P}$ in Step (ii), and assigns the other jobs to $\mathcal{S}$. Therefore, its cost is

$$\frac{Mk(m/(\gamma k)+\varepsilon) + mM}{m}.$$

There is a feasible schedule which assigns the second part of the jobs to $\mathcal{P}$ and the third part to $\mathcal{S}$, therefore, the optimal cost is no more than $M + m/(\gamma k) + 2\varepsilon$. As M tends to ∞ and ε tends to 0, the ratio of these costs tends to $1 + 1/\gamma$, hence, we proved that the third bound is tight.

□

To find the best values of α, γ, we have to choose $\alpha = \gamma = 1/\sqrt{2}$. By substituting these values into Theorem 2, we obtain the following result.

Corollary 4 *Algorithm $A(1/\sqrt{2}, 1/\sqrt{2})$ is $1 + \sqrt{2}$-competitive on LS.*

4 Lower bounds

In this section we present the following lower bounds.

Theorem 5 *Let (k, m) be an arbitrary pair of positive integers. If an online algorithm is c-competitive for the $LS(k, m)$ problem, then $c \geq (1 + \sqrt{5})/2 \approx 1.618$.*

Proof. We prove this statement by contradiction. Suppose there is a pair (k, m) and an algorithm A which is $c < (1 + \sqrt{5})/2$-competitive for the problem $LS(k, m)$. Consider the following list L of jobs. The first part contains k jobs of size $((\sqrt{5} - 1)/2, 1)$, and the next part contains k jobs of size $(1, \infty)$. In this case, the optimal offline algorithm assigns the first k jobs to $\mathcal{S}$, and the next k jobs to $\mathcal{P}$, hence $OPT(L) = 1$. On the other hand, since A is c-competitive it must assign the first k jobs to $\mathcal{P}$, otherwise, we omit the next k jobs, and the offline optimum is $(\sqrt{5} - 1)/2$, while the cost of the algorithm is 1. Therefore, the online algorithm must assign all jobs to $\mathcal{P}$, and thus, it has a makespan $(1 + \sqrt{5})/2$. This yields that $A(L)/OPT(L) = (1 + \sqrt{5})/2$, which is a contradiction.

□

We can obtain a sharper, general lower bound as follows.

Theorem 6 *Let k be a fixed constant. If an online algorithm is c-competitive for every m on the problem $LS(k, m)$, then $c \geq (3 + \sqrt{17})/4 \approx 1.781$*

Proof. We prove this statement by contradiction. Suppose that there exist such k and an online algorithm A, that A is c-competitive for every m on the problem $LS(m, k)$, where $c < (3 + \sqrt{17})/4$. For the sake of simplicity, in the rest of the proof we denote the number $(3 + \sqrt{17})/4$ by b. Let m be greater than $5k$ and consider the following sequence of jobs. The first part contains k jobs of size $(1/b, 1)$, and the following $m - k$ jobs have size $(\frac{1}{m-k}, 1)$. Finally, depending on the decisions made by A, we finish the sequence with k jobs of size $(1, \infty)$, this is list L_1, or we finish the list with $m - k$ jobs of size $(\infty, 1)$, this is list L_2.

Consider first the offline optimum. In the first case we can assign the first m jobs to $\mathcal{S}$ and the last k jobs to $\mathcal{P}$, this schedule has cost 1. In the second case,

we can assign the first k jobs and the last $m-k$ jobs to $\mathcal{S}$, and the other $m-k$ jobs to $\mathcal{P}$, and thus, we can obtain a makespan of 1. Therefore, $OPT(L_1) \leq 1$, and $OPT(L_2) \leq 1$.

Consider the algorithm A. Since A is an online algorithm, it cannot see any difference between L_1 and L_2 before it gets the $m+1$-th job, and thus, it has the same behaviour in both cases. Furthermore, A is c-competitive with $c < b$, therefore it must assign the first k jobs to $\mathcal{P}$, otherwise we get a contradiction in the same way as in the proof of Theorem 5. From the next $m-k$ jobs A can assign x to $\mathcal{P}$ and $m-k-x$ to $\mathcal{S}$. Therefore, in the case of list L_1 we have that $A(L_1) \geq \frac{1}{b} + \frac{x}{m-k} + 1$ and in the case of list L_2 we get that $A(L_2) \geq \frac{2m-2k-x}{m}$. The algorithm can choose x to be any integer between 1 and $m-k$, and we can choose the list which yields the greater makespan. Therefore, since the offline optimum is at most 1 for both lists and A is c-competitive, we have that

$$c \geq \min_{1 \leq x \leq m-k} \max\{\frac{1}{b} + \frac{x}{m-k} + 1, \frac{2m-2k-x}{m}\}.$$

Here we omitted the condition that x is an integer. This does not cause any problem since it decreases the right side of the inequality. It can easily be seen that the function of x, which is on the right side of the inequality, is minimal for

$$x = \frac{m(m-k)}{2m-k}(\frac{m-2k}{m} - \frac{1}{b}).$$

If we substitute this value into the bound for c, we obtain that

$$c \geq \frac{1}{b} + \frac{3m-3k}{2m-k} - \frac{m}{(2m-k)b}.$$

Since this inequality is valid for arbitrary m, it is also valid if we let m to tend to infinity. Therefore,

$$c \geq \frac{3}{2} + \frac{1}{2b}.$$

On the other hand, $b = \frac{3}{2} + \frac{1}{2b}$, and thus, we obtain that $c \geq b$ which is a contradiction.

□

5 Conclusions

In this paper we investigated a particular scheduling problem on a two-layer multiprocessor architecture. We showed that the greedy approach works well only in the cases where the layers contain around the same number of machines. We also presented a better algorithm for the general case, which has a constant competitive ratio for arbitrary number of machines. It was also proved that there exists no online algorithm with smaller competitive ratio than 1.781.

In relation with the problem considered, some further questions arise.

Concerning the *LS* problem there is a gap between our lower and upper bounds. It would be nice to decrease this gap by finding more efficient algorithms or better lower bounds.

In the problem considered the objective is the maximum of the two makespans, which function is the l_∞ norm of the vector constructed from the makespans. In [5] the l_1 norm is investigated. It would be also interesting to study other l_p norms.

We considered the problem with two sets of machines. A straightforward generalization is to consider a problem with more sets of machines.

References

[1] S. Albers, Better Bounds for Online Scheduling, *SIAM Journal of Computing*, 29 (1999) 459–473.

[2] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, O. Waarts, On-line load balancing with applications to machine scheduling and virtual circuit routing, *J. of the ACM*, 44 (1997) 486–504.

[3] Y. Bartal, S. Leonardi, A. Marchetti-Spaccamela, J. Sgall and L. Stougie, Multiprocessor scheduling with rejection, *SIAM Journal of Discrete Mathematics*, 13 (2000) 64–78.

[4] R. L. Graham, Bounds for certain multiprocessor anomalies, *Bell System Technical Journal*, 45 (1966) 1563–1581.

[5] Cs. Imreh, Online classification with offline scheduling, *Algoritmica*, submitted for publication.

[6] S. A. Roosta, *Parallel Processing and Parallel Algorithms*, Springer-Verlag, New York, 1999.

[7] S. S. Seiden, Preemptive Multiprocessor Scheduling with Rejection, *Theoret. Comput. Sci.*, to appear.

[8] Sgall J. On-line scheduling, In *Online algorithms: The State of the Art*, Lecture Notes in Computer Science, Vol. 1442, G. Woeginger and A. Fiat (Eds.), Springer-Verlag, 1998, 196–231.

Acta Cybernetica **15** (2001) 173–184.

Parallel Simulation of Spiral Waves in Reacting and Diffusing Media

E. M. Ortigosa*, L. F. Romero* and J. I. Ramos*

Abstract

The propagation of the spiral waves in excitable media is governed by the non-linear reaction-diffusion equations. In order to solve these equations in the three-dimensional space, two methods have been implemented and parallelized on both shared- and distributed- memory computers. These implicit methods linearize the equations in time, following alternate directions in the first case (ADI), and using the Crank-Nicolson discretization in the second case. A linear system of algebraic equations has been obtained and it has been solved using direct methods in the ADI technique, while in the second case has been used the conjugated gradient (CG) method. An optimized version of the CG algorithm is presented here, in which the largest efficiency has been obtained.

1 Introduction

Reaction-diffusion equations are ubiquitous in biology, combustion, ecology, etc., because of their relevance in pattern formation, ignition and extinction phenomena, etc. [1-3]. Many studies of these equations are related with equations for activators and inhibitors in one or two spatial dimensions, e.g. the Belousov-Zhabotinskii, Brusselator and Oregonator models, with and without extinction [3]. Of special interest to the study presented in this paper are the analytical and numerical analyses of the propagation of spiral waves in two-dimensional domains, where it has been observed that these waves have a periodic pattern in the absence of heterogeneities. They may exhibit breathing motions in the presence of obstacles or may simply be extinguished by means of the activation of a control parameter in a sufficiently large region of the computational domain. In non-homogeneous media, spiral waves are characterized by steep gradients in space and relaxation-type oscillations whose accurate simulation demands small spatial and temporal steps.

In three dimensions, there have been very few analytical and numerical studies of spiral waves, presumably because of both the large difficulties in examining wave propagation in three-dimensional space and the cost of such simulations [4]. As a

*Departamento de Arquitectura de Computadores, Campus de Teatinos, Universidad de Málaga, E-29071, Spain, e-mail: eva,felipe@ac.uma.es

consequence, filament models based on Fréchet formulae and differential geometry have been developed; these models are analogous to those of vortex filaments in theoretical fluid mechanics.

In this work, three-dimensional simulations of the propagation of spiral waves in a cubic volume, without obstacles, and in the presence and absence of extinction sources are presented. These numerical simulations have been carried out by means of both time linearized and non-linearized techniques with and without operator splitting, i.e., with and without approximate factorization of the three-dimensional operator into one-dimensional ones. The numerical methods employed in the discretization of the governing partial differential equations have been implemented in a parallel fashion in both shared- and distributed-memory computers, and their performance is reported in this paper.

2 Governing equations

Consider the following system of reaction-diffusion equations (Belousov-Zhabotinskii model):

$$\frac{\partial \mathbf{U}}{\partial t} = \mathbf{D}\left(\frac{\partial^2 \mathbf{U}}{\partial x^2} + \frac{\partial^2 \mathbf{U}}{\partial y^2} + \frac{\partial^2 \mathbf{U}}{\partial z^2}\right) + \mathbf{S}(\mathbf{U}). \tag{1}$$

where $\mathbf{D}$ is a diagonal diffusivity tensor, $\mathbf{U} = (u, v)^T$, t is time, x, y and z denote spatial coordinates, $\mathbf{S}$ is a non-linear term,

$$\mathbf{S} = \left(\frac{u(u-1) - (fv + \phi)\frac{u-q}{u+q}}{\epsilon}, u - v\right)^T, \tag{2}$$

T denotes transpose, f, q and ϵ are constants, and ϕ is a control parameter which can be a function of the space and/or time. The equation (1) has been solved in a cube of side equal to 15 non-dimensional units. Unless otherwise stated, $f = 1.4$, $q = 0.002$, $\epsilon = 0.01$, $\phi = 0$ and the diagonal terms of the diffusivity tensor are $d_u = 1$ and $d_v = 0.6$.

Discretizing the time variable in equation (1) by means of a Crank-Nicolson method, one can obtain a non-linear elliptic equation, which can be solved at each time step. The second-order spatial derivatives were discretized by means of three-point, second order accurate finite difference methods that provide a large system of algebraic equations at each time step. The Newton-Raphson method was employed to solve the resulting non-linear system of algebraic equations. If a single iteration of Newton-Raphson method is used, this method is known as Time-Linearization method.

In addition to these techniques, an approximate factorization of the three-dimensional operator into a sequence of one-dimensional ones (here referred to as ADI) was also used. This technique reduces the solution of the linear elliptic equation in three dimensions to the solution of one-dimensional, linear, two-point boundary value problems in each spatial dimension, but introduces second-order

(in time) approximate factorization errors which can be eliminated in an iterative way. On the other hand, the approximate factorization errors are of the same order of magnitude as those introduced by the time discretization but may be large where the norm of the Jacobian matrix of the source terms is large, i.e., at the edges of the spiral wave.

In the next section some details about these methods are presented.

2.1 ADI method

The discrete operators corresponding to equation (1) can be written as

$$\begin{aligned}
\left(\mathbf{I} - \frac{k}{2\Delta x^2}\mathbf{D}\delta_x^2 - \frac{1}{2}k\mu_1\mathbf{J}^n\right)\Delta\mathbf{U}^{**} = \\
\mathbf{D}\left(\frac{k}{\Delta x^2}\delta_x^2 + \frac{k}{\Delta y^2}\delta_y^2 + \frac{k}{\Delta z^2}\delta_z^2\right)\mathbf{U}^n + k\mathbf{S}^n, \\
\left(\mathbf{I} - \frac{k}{2\Delta y^2}\mathbf{D}\delta_y^2 - \frac{1}{2}k\mu_2\mathbf{J}\right)\Delta\mathbf{U}^{*} = \Delta\mathbf{U}^{**}, \\
\left(\mathbf{I} - \frac{k}{2\Delta z^2}\mathbf{D}\delta_z^2 - \frac{1}{2}k\mu_3\mathbf{J}\right)\Delta\mathbf{U} = \Delta\mathbf{U}^{*},
\end{aligned} \tag{3}$$

where the approximate factorization errors have been neglected, k is the time step, Δx is the spatial step size in the x direction, $0 \leq \mu_i \leq 1$, $i = 1, 2, 3$, $\mu_1 + \mu_2 + \mu_3 = 1$, the superscript n denotes the n-th time level, $\mathbf{J}$ denotes the Jacobian matrix of the mapping $\mathbf{U} \rightarrow \mathbf{S}(\mathbf{U})$, $\delta_x^2 v_i = v_{i+1} - 2v_i + v_{i-1}$, and $\Delta\mathbf{U} = \mathbf{U}^{n+1} - \mathbf{U}^n$.

2.2 Crank-Nicolson methods

If the system of equations (1) is solved by means of a Crank-Nicolson method and a time linearization, one linear system of algebraic equations is obtained.

$$\begin{aligned}
\left(\mathbf{I} - \mathbf{D}\left(\frac{k}{2\Delta x^2}\delta_x^2 + \frac{k}{2\Delta y^2}\delta_y^2 + \frac{k}{2\Delta z^2}\delta_z^2\right) - \frac{1}{2}k\mathbf{J}^n\right)\Delta\mathbf{U} = \\
\mathbf{D}\left(\frac{k}{\Delta x^2}\delta_x^2 + \frac{k}{\Delta y^2}\delta_y^2 + \frac{k}{\Delta z^2}\delta_z^2\right)\mathbf{U}^n + k\mathbf{S}^n.
\end{aligned} \tag{4}$$

A comparison between the numerical results obtained with equations (3) and (4), will indicate the magnitude of the approximate factorization errors of the ADI method. The (iterative) conjugate gradient method have been used for the resolution of the system (4), because of the magnitude and dispersion of the system.

The choice of a good preconditioner for the coefficients matrix is the most important factor that influence on the speed of convergence of the CG. We have tested the Jacobi (J), block Jacobi (BJ), incomplete Cholesky (IC) and incomplete-block

Cholesky (IBC) preconditioners in our studies. In the first and third preconditioners, two dependent variables per node have been used to form 2x2 blocks; calculations have also been performed without any preconditioner (NP), but the number of iterations of the CG method becomes extremely large and can be more expensive than a direct method. In Table 1, the average number of iterations required by each method to converge is presented as a function of the grid size; the values shown in this table correspond to $k = 0.0004$ and 20 time steps (n.a.: not available).

Table 1: Average number of iterations of CG for convergence.

Grid	NP	J	BJ	IC	IBC
51^3	5.95	2.25	2.25	2.3	2.20
101^3	6.00	3.1	3.20	3.2	3.1
201^3	n.a.	5.05	5.15	5.2	n.a.

The above table shows that the incomplete-block Cholesky factorization is the most efficient preconditioner; however, the cost associated with this incomplete factorization is larger than the associated with the decrease in the number of iterations[1]. For these reasons, a Jacobi preconditioner has been used in all the simulations presented below.

Another main issue when solving linear systems of algebraic equations is the ordering of the equations. There are two criteria for ordering: the original differential order equation and the grid point where the discretization takes place. Depending on the approach selected, two different orderings are obtained, named blocking of the equations and blocking of the variables respectively. Blocking of the equations results in a small number (2 in our problem) of very large blocks, while blocking of the variables per node results in more small blocks (2×2). Here, we have tested both types of blocking and found that blocking of the variables results in faster simulations due mainly to a 3% and 5% reduction in primary and secondary cache misses respectively.

3 Parallel implementation

3.1 Parallelization of ADI

This technique has been implemented on an Origin 2000, using a shared memory model (*openMP* libraries). The dynamic block cartesian decomposition (DBCD) has been used for parallel implementation. In this technique, each processor has a contiguous grid block, but the decomposition changes as the one-dimensional

[1] In order to decrease the factorization cost, calculations have been performed using a frozen preconditioner for several time steps. Results showed that the number of iterations of the CG method increase substantially due to the large relational speeds of the spiral wavers considered in this paper.

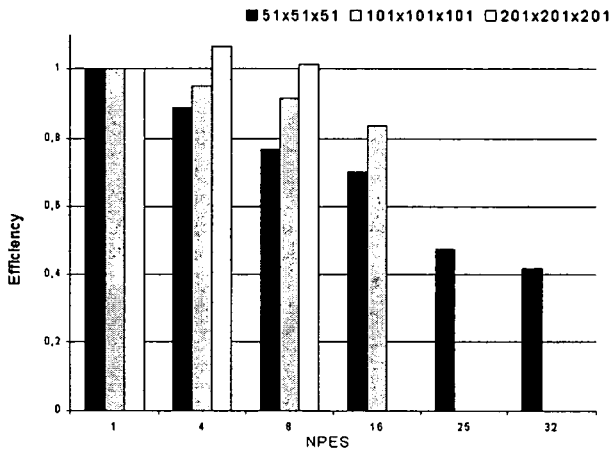


Figure 1: Efficiency of the ADI method as a function of the number of processors (NPES) and the grid size.

operator is changed, i.e., it changes according to the spatial direction. As stated in [5], the optimal solution is reached when the data is partitioned in the z-direction while solving the algebraic equations in the x- and y-directions, whereas a partition in the x- or y-directions may be employed when solving the algebraic equations in the z-direction. The main drawback of this technique is that a lot of accesses are required to remote data before the solution in the z direction is obtained. This produces a high number of cache misses.

A possible alternative is the Bruno-Capello algorithm [5] that partitions the domain in blocks of $\sqrt{n} \times \sqrt{n} \times \sqrt{n}$ in such a manner that there is no more than one block per processor in each plane coordinate; however, this algorithm imposes some limitations to the parallel system because of its requirements for a number of processors equal to the square of a natural number.

Results in Figure 1 indicate that the parallel efficiency of DBCD is quite close to 1, even with a high number of processors. This ideal behavior is owed in great measure to the regularity of the data, and the consequent efficiency of the compiler in the inclusion of directives for the premature search of cache blocks (*prefetching*). Moreover, as will be illustrated bellow, the main drawback of the ADI method is its sequential execution time compared with that of the Crank-Nicolson method.

3.2 Parallelization of Crank-Nicolson

First, the implementation of Crank-Nicolson method combined with the CG (CN-CG) method has been performed using a shared-memory model. In each time step it is necessary to generate the coefficients matrix and the independent term vector

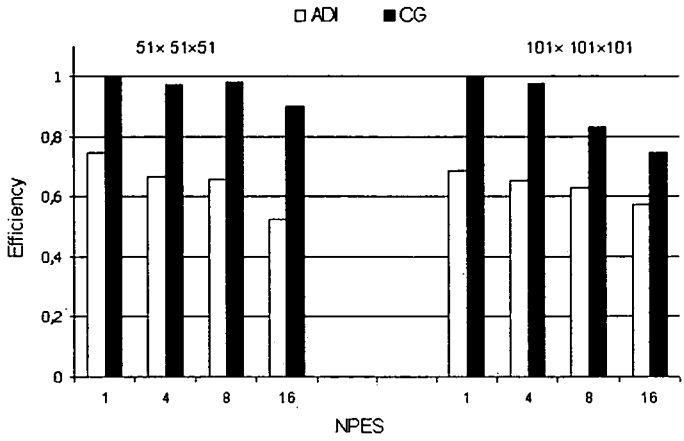


Figure 2: Efficiency of the ADI and CG methods as a function of the number of processors and the grid size.

(rhs), and to solve the system. In all these stages we have used a data-parallel model based on a grid partition along the z axis that coincides with a block partition of both the matrix and the vector rhs.

The parallelization of the CG method is very simple since every vector operation of this algorithm can be parallelized separately, i.e., each processor executes scalar operations on a subset of the components of a vector, and a block partition of the vectors is sufficient to obtain a good performance.

The CG method contains two inner products, three $\alpha\mathbf{x} + \mathbf{y}$ operations and one matrix vector product whose computations require at least three synchronization points, and a communication step in order to obtain the final result. It is not possible to merge messages and those operations imposes severe limitations to the parallel efficiency of the CG algorithm. However, an optimized rearrangement of CG for parallel computations can be found in [6]. Unfortunately, this optimized method is only useful when an incomplete factorization is employed as a preconditioner. As it have been shown above, this kind of preconditioner does not result in any acceleration in the convergence of the problem considered here.

On the other hand, another methods based on the CG not requiring the use of incomplete preconditioner have been described. In particular we have considered a model (Aykanat, Özgüner, F. and Scott, D.S. [7], also Chronopoulos and Gear [8]) that reduces the number of synchronization points and reduction messages to one per iteration, performing $2n$ additional floating operations. In this method we have introduced automatic *prefetching*, in order to reduce the cost produced by the accesses to remote memory. Experiments with this modified CG method have shown that there is only a small increase in the sequential computational time but its parallel performance is excellent.

Figure 2 shows the efficiency of ADI and CG as a function of both the number of processors and the grid size using a shared-memory model, and indicates that the efficiency of ADI is smaller than that of CG, although the differences decrease, as the grid is refined. However, the fact that the computational time for the sequential version is smaller for the CG method, has been critical in the decision to continue our work with the Crank-Nicolson-CG model.

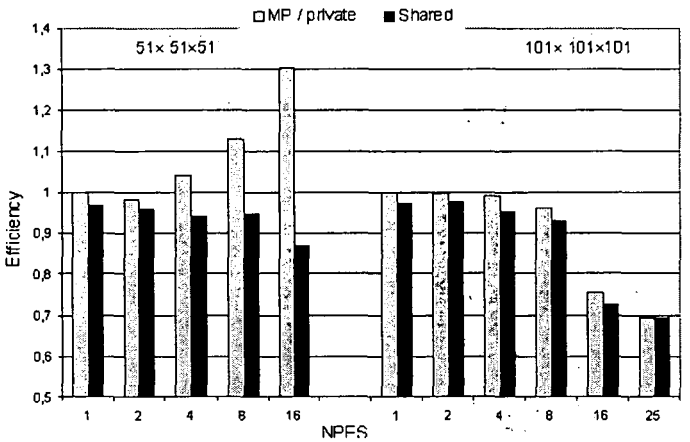


Figure 3: Efficiency of the CG methods as a function of the number of processors with shared-memory and message-passing models with private memory.

4 Optimization of Crank-Nicolson method

One of the main aims of this work is to perform a comparison between accessing in advance to cache blocks using *prefetching* and using asynchronous messages in the message-passing model (overlapped with computations) on shared- and distributed-memory computers. Therefore, we have implemented the conjugated gradient method on an Origin2000 computer, using both private memory and the *shmem* libraries for communications.

In Figure 3 we compare the efficiency for the two parallel implementations of the CG, being better for the message passing model. In the best case, an efficiency of 130% was obtained using 16 processors in a 51 point grid.

This result can be justified by the fact that the communication of large blocks of data is more efficient than having many cache misses in a shared memory model.

This difference becomes noticeable in thick grids where the access to remote memories is more frequent, since the relationship communications ($O(n^2)$)/computations ($O(n^3)$) is higher. Figure 3 also shows that the efficiency of both implementations of the CG, for the 101 point grid, in a shared memory

computer (Origin2000) is degraded as the grid size increases, because in these cases the capacity of the cache memory is insufficient for all the data.

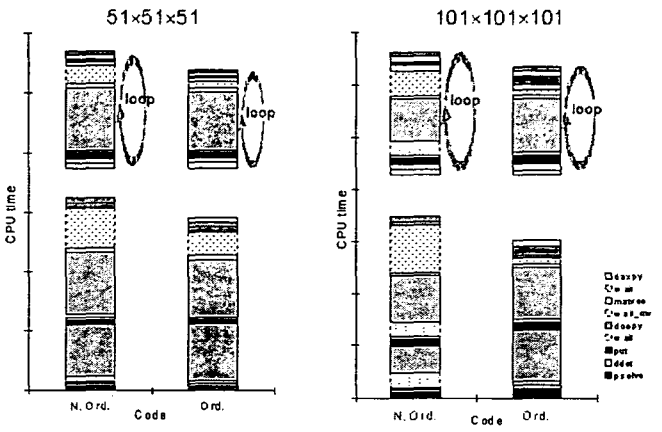


Figure 4: CPU times per processor for the non ordering and ordering codes.

An important performance loss can be observed for the $101 \times 101 \times 101$ grid when the number of processors is very high. Although this loss can be justified by the high cost of the remote accesses in proportion to the computations that each processors performs, we consider very important for our problem to obtain good performance under such conditions (very fine grids and higher number of processors). A detailed analysis (of the origin of this performance loss) shows that, for example, with a 101 point grid and 16 processors, almost the 30% of the execution time in message-passing the processors are waiting for the data. Therefore, the sending of asynchronous messages does not produce the expected results, because, in the most of cases, the sent data are required immediately in the following operation, and there are no intermediate computations that hide the latency of these messages.

The conjugate gradient algorithm of Gear et al., has been modified based on a reordering heuristic of the computations and communications [9]. We have moved several calculations to those places where the latencies of the messages can be hidden and the delays have been almost eliminated. This algorithm is based mainly on delaying the calculation of the solution, $\Delta \mathbf{U}$ (equation 4), from each iteration to the following one. When the convergence is reached, an update of $\Delta \mathbf{U}$ (to obtain the solution of the system) is required. This update is performed directly on the solution $\mathbf{U}$. After the conjugate gradient, a message with the bound values of $\mathbf{U}$ is sent to the neighbor processors and is overlapped with the update of $\Delta \mathbf{U}$. Figure 4 shows the execution time for each processor for each conjugated gradient routines (non ordered and ordered codes), as well as the latency time (dotted area). The bottom part of the graphic corresponds to the initialization part of the iterative

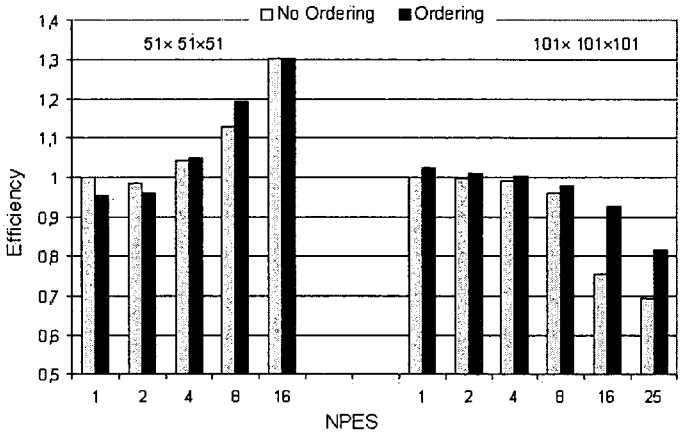


Figure 5: Efficiency comparison between the non-ordering and ordering codes.

algorithm, and the upper part corresponds to the inner loop (therefore it has to be repeated according to the number of iterations). In Figure 5 it can be seen how the reordering allows to reduce significantly the latency time, decreasing about a 15% the CPU time and increasing in an equivalent way the efficiency. This improvement is higher for a 101 point grid, where the communications cost is very important when the number of processors increases.

5 Physical Results

Calculations have been performed 1) without extinction, i.e., $\phi = 0$, 2) with an extinction barrier, i.e., $\phi = 0.2$ for $-0.3 \leq y \leq 0.3$ and $\phi = 0$ otherwise, and 3) with a localized extinction zone which is a cube with center located at $(0, 0, 0)$ and length equal to 1.65 where $\phi = 0.2$. In all cases, homogeneous Neumann boundary conditions were employed at the faces of the computational domain, $k = 0.0004$, and the calculations were performed until $t = 60$, starting from initial conditions corresponding to a wedge for u. In all the calculations considered in this paper, it has been observed that the solution became periodic at about $t = 25, 30$, and 28 for the first, second and third cases, respectively. In the first case, i.e., without extinction, it has been shown that the spiral wave rotates around an axis parallel to the z-direction which passes through the center of the computational domain and remains anchored there, the u solution is symmetric about the $z = 0$ plane, and the rotational speed of the spiral wave decreases as it approaches the planes $z = -7.5$ and $z = 7.5$. Some sample results illustrating the u solution at $t = 41.6$ are presented in Figure 6 which indicates that, at this time, the spiral wave is nearly absent near the bottom and top planes, whereas the influence of the initial conditions can still be observed near these planes. The u-solution also shows that at

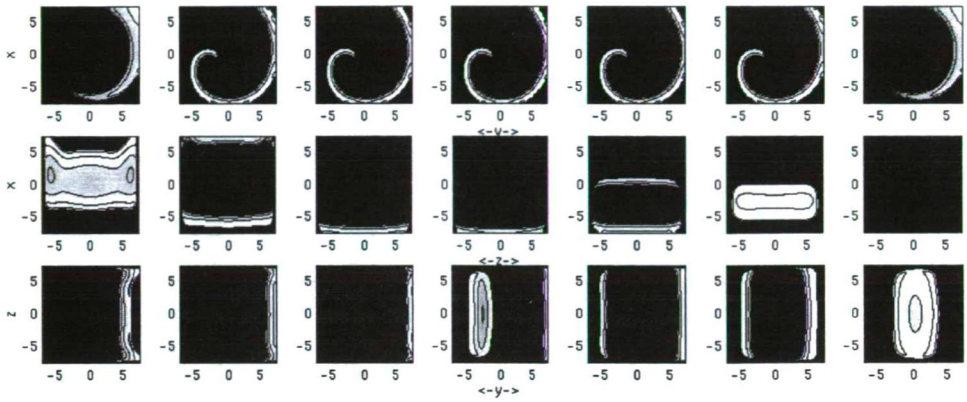


Figure 6: u-solution for a 101×101×101 grid at $t = 41.6$. From left to right: a) top: k=100, 84, 68, 51, 34, 18, 2, b) middle: j=100,84, 68, 51, 34, 18, 2, c) bottom: i=100, 84, 68, 51, 34, 18, 2.

$t = 41.6$, near to the top and bottom planar boundaries, the spiral wave is thicker and much shorter than that near $z = 0$.

In the second case, the extinction barrier is a slab parallel to the $x - z$ plane where the value of ϕ is different from zero; therefore, the source term for u decreases exponentially whenever $u > q$ by an amount that is proportional to ϕ, i.e., the spiral wave may be extinguished as indicated in Figure 7. The u- solution in $x = constant$ planes indicates that almost planar fronts propagate initially from the left to the right boundaries but, on encountering the extinction barrier, they decelerate and emerge from this barrier until they reach the right boundary.

In the third case, the results are similar to those of the first one except for the fact that the tip of the spiral wave rotates around the extinction source, and the wave is shorter and has a thicker tip. In the first case, the tip is anchored on the vertical axis passing through the center of the cube, while, in the third one, the anchoring point describes a trajectory which corresponds to the projection of the extinction cube into the $x - y$ plane.

6 Conclusions

Two numerical methods (ADI and Crank-Nicolson) for the numerical solution of three-dimensional reaction-diffusion equations corresponding to spiral wave propagation in excitable media have been parallelized in shared- and distributed-memory computer. The parallelization of the approximate factorization technique has been carried out with a dynamic block cartesian decomposition (DBCD) and its efficiency is quite close to one, even with a high number of processors. The parallelization of the Crank-Nicolson method has been performed by means of an optimization of the conjugate gradient method where the latency times have been almost elimi-

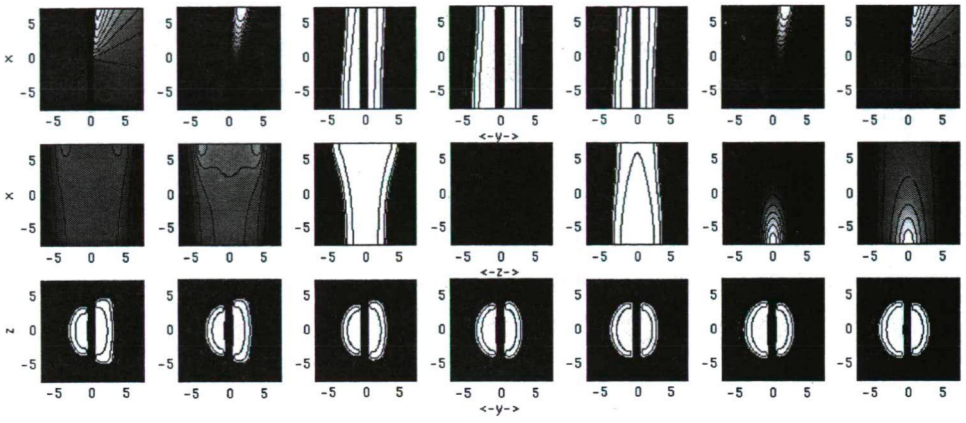


Figure 7: u-solution for a 101×101×101 grid at $t = 41.6$. From left to right: a) top: k=100, 84, 68, 51, 34, 18, 2, b) middle: j=100, 84, 68, 51, 34, 18, 2, c) bottom: i=100, 84, 68, 51, 34, 18, 2.

nated in a message-passing programming model. As a result, we obtain efficiencies close to the ideal, even with a higher number of processors. The scalability of our model should allow maintenance of the efficiency with proportional increments of the problem size and the number of processors.

This work has its natural continuation in the implementation of the mentioned optimization in a shared-memory environment, where the cost of remote accesses will decrease using the manual inclusion of *prefetching* directives. The objectives of a future work will be a comparison of both models and a extrapolation to the ADI method of these ideas.

References

[1] Murray, J.D. *Mathematical Biology*, Springer-Verlag: New York, 1989.

[2] Williams, F.A. *Combustion Theory*, second edition, Addison-Wesley: New York, 1985.

[3] Holden, A.V., Markus, M. and Othmer, H.G. (eds). *Nonlinear Wave Propagation in Excitable Media*, Plenum Press: New York, 1991.

[4] Keener, J. and Sneyd, J. *Mathematical Physiology*, Springer-Verlag: New York, 1998.

[5] Van der Wijngaart, R.F. Efficient Implementation of a 3-Dimensional ADI Method on the iPSC/860, *Supercomputing'93*, pp. 102–111, 1993.

[6] Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. and Van de Vorst, H., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM: Philadelphia, 1994.

[7] Aykanat, C., Özgüner, F. and Scott, D.S. Vectorization and parallelization of the conjugate gradient algorithm on hypercube-connected vector processors, *Microprocess. Microprogram.*, 29, pp. 67–82, 1990.

[8] Chronopoulos, A. T. and Gear C. W. S-step iterative methods for symmetric linear systems, *J. comput. Appl. Math.*, 25, pp. 153–168, 1989.

[9] Romero, L.F., Ortigosa, E.M. and Ramos, J.I. Parallel Strategies for the VMEC Program, *J. Parallel Computing*, to appear.

Acta Cybernetica **15** (2001) 185–200.

Generation and Reconstruction of hv-convex 8-connected Discrete Sets

Emese Balogh*

Abstract

An algorithm is given to generate 2-dimensional *hv*-convex 8-connected discrete sets uniformly. This algorithm is based on an extension of a theory previously used for a more special class of *hv*-convex discrete sets. The second part of the paper deals with the reconstruction of *hv*-convex 8-connected discrete sets. The main idea of this algorithm is to rewrite the whole reconstruction problem as a 2SAT problem. Using some a priori knowledge we reduced the number of iterations and the number of clauses in the 2SAT expression which results in reduction of execution time.

Keywords: Discrete tomography; Reconstruction from projections; Convex discrete set; Generation at random;

1 Introduction

One of the most important problems of *Discrete Tomography* is the reconstruction of 2-dimensional discrete sets from their two orthogonal projections. Often the reconstructed object should fulfil some additional properties including connectivity or convexity. In certain classes, the reconstruction is NP-hard [8, 19], therefore, the most frequently studied classes are those, where the reconstruction can be performed in polynomial time, like the *hv*-convex polyominoes and *hv*-convex 8-connected discrete sets. In this paper we present an algorithm for the reconstruction in the class of *hv*-convex 8-connected discrete sets.

Chrobak and Dürr [5] showed that the reconstruction of an *hv*-convex polyomino is equivalent to the evaluation of a suitable constructed 2SAT expression. This method was extended by Kuba [16] for the class of *hv*-convex 8-connected discrete sets. In this paper we give the description of a modified algorithm following the same idea as Chrobak and Dürr and Kuba for the class of *hv*-convex 8-connected discrete sets. We reduced the number of cases and modified the 2SAT expression introducing some preliminary information concerning the object which are obtained from the two orthogonal projections. The class of 8-connected discrete sets includes

*Department of Informatics, University of Szeged, H-6701 Szeged, PO. Box 652, Hungary, e-mail: bmse@inf.u-szeged.hu

the class of convex polyominoes. It has applications in image processing, biplane angiography, electron microscopy, and others.

In Section 2 we introduce some basic definitions and notations. To test the reconstruction algorithm we need to generate such sets with uniform distribution. In [14] Hochstättler, Loebl, and Moll presented an algorithm for the class of *hv*-convex polyominoes. In Section 3 we extend the more general class of *hv*-convex 8-connected discrete sets. Finally, in Section 4 we present the reconstruction algorithm.

2 Definitions and Notations

Let $\mathbb{Z}^2$ denote the 2-dimensional integer lattice, its elements can be represented by unit squares called cells. The finite subsets of $\mathbb{Z}^2$ are called discrete sets. Let S be a discrete set. Then there is a discrete rectangle Q of size $m \times n$ where m and n are positive integers, $Q = \{1, \ldots, m\} \times \{1, \ldots, n\}$, such that Q is the smallest discrete rectangle containing S. The number of cells of S is called the *area* of S. The *perimeter* of S is the number of pairs of cells sharing a common side, where one cell is an element of S and the other one is not. Note that if $Q = \{1, \ldots, m\} \times \{1, \ldots, n\}$ is the smallest discrete rectangle containing S then the perimeter of S is $2m + 2n$.

The discrete set can be represented as a binary matrix $(s_{ij})_{m \times n}, s_{ij} \in \{0, 1\}$, such that $s_{ij} = 1$ if $(i, j) \in S$, and $s_{ij} = 0$, otherwise.

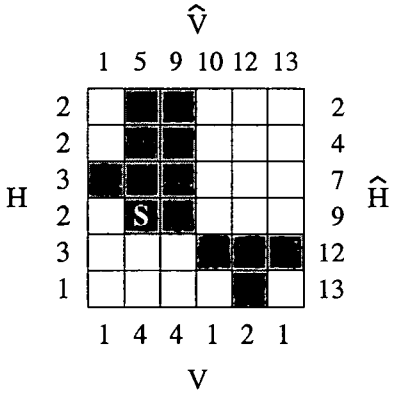


Figure 1: An *hv*-convex 8-connected discrete set S, its elements are marked by dark grey squares. H and V are the row and column sum vectors of S. The cumulated sums of H and V are denoted by $\widehat{H}$ and $\widehat{V}$ (their first elements, $\widehat{h}_0$ and $\widehat{v}_0$ are not indicated in the figure).

Let $\mathbb{N}$ denote the set of positive integers. For any discrete set S we define its *projections* by the operations $\mathcal{H}$ and $\mathcal{V}$ as follows. $\mathcal{H} : \mathcal{S} \longrightarrow \mathbb{N}^m$, $\mathcal{H}(S) = H = (h_1, \ldots, h_m)$ where $h_i = \sum_{j=1}^{n} s_{ij}, i = 1, \ldots, m$, and $\mathcal{V} : \mathcal{S} \longrightarrow \mathbb{N}^n$, $\mathcal{V}(S) = V = (v_1, \ldots, v_n)$ where $v_j = \sum_{i=1}^{m} s_{ij}, j = 1, \ldots, n$. The vectors H and V are

called the *projections* or *row* and *column sum vectors* of S, respectively (see Fig 1). The cumulated vectors of H and V are denoted by $\widehat{H} = (\widehat{h}_0, \widehat{h}_1, \ldots, \widehat{h}_m)$ and $\widehat{V} = (\widehat{v}_0, \widehat{v}_1, \ldots, \widehat{v}_n)$, that is, $\widehat{h}_0 = 0, \quad \widehat{h}_i = \widehat{h}_{i-1} + h_i, \quad i = 1, \ldots, m$, and $\widehat{v}_0 = 0, \quad \widehat{v}_j = \widehat{v}_{j-1} + v_j, \quad j = 1, \ldots, n$ (see Fig. 1). Let $T = \sum_{i=1}^{m} h_i = \sum_{j=1}^{n} v_j$. Let S and S' be discrete sets.

We say that S and S' are *tomographically equivalent* (w.r.t. the row and column sum vectors) if $\mathcal{H}(S) = \mathcal{H}(S')$ and $\mathcal{V}(S) = \mathcal{V}(S')$.

The *4-neighbours* of a cell $(i,j) \in \mathbb{Z}^2$ are $(i-1,j)$, $(i,j-1)$, $(i,j+1)$, $(i+1,j)$ and the cell (i,j) itself. The *8-neighbours* of a cell $(i,j) \in \mathbb{Z}^2$ are the 4-neighbours and $(i-1,j-1)$, $(i-1,j+1)$, $(i+1,j-1)$, $(i+1,j+1)$. The sequence of distinct cells $(i_0,j_0),\ldots,(i_k,j_k)$ is a *4-path/8-path* from cell (i_0,j_0) to cell (i_k,j_k) in a discrete set S if each cell of the sequence is in S and (i_l,j_l) is 4-adjacent/8-adjacent, respectively, to (i_{l-1},j_{l-1}) for each $l = 1,\ldots,k$. Two points are *4-connected/8-connected* in the discrete set S if there is a 4-path/8-path, respectively, in S between them. A discrete set S is *4-connected/8-connected* if any two points in S are *4-connected/8-connected*, respectively. The 4-connected set is also called *polyomino*. From the definitions it follows that the class of 4-connected sets is a subset of the class of 8-connected sets (see Fig. 2).

The discrete set S is *horizontally convex* (or shortly, *h-convex*) if its rows are 4-connected. Similarly, a discrete set S is *vertically convex* (or, shortly, *v-convex*) if its columns are 4-connected. If a discrete set is both h-convex and v-convex then it is called *hv*-convex.

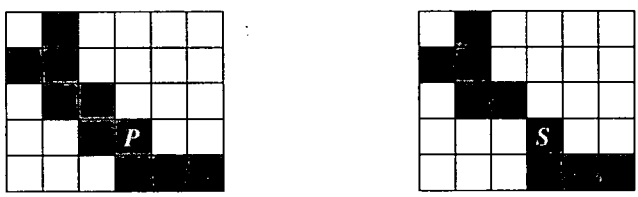


Figure 2: (a) P is a 4-connected *hv*-convex discrete set. (b) S is an 8-connected but not 4-connected *hv*-convex discrete set.

3 Generation of *hv*-convex 8-connected Discrete Sets at Random

Hochstättler, Loebl, and Moll [14] gave an algorithm for the generation of *hv*-convex polyominoes. In this paper, we extend this algorithm to the more general class of *hv*-convex 8-connected sets. First we describe the algorithm for calculating the number of *hv*-convex 8-connected discrete sets with fixed perimeter which is greater than the number of *hv*-convex 4-connected discrete sets with the same perimeter. Then using the same method as presented in [14] we construct a bijection between a

set of *hv*-convex 8-connected discrete sets of a given perimeter and a finite interval of natural numbers will be given. This bijection allows us to generate such sets at random with uniform distribution in polynomial time. We use almost the same notation as in [14].

3.1 Definitions

A *strip* is a discrete rectangle of height 1. Let $L(s)$ and $R(s)$ denote the leftmost and rightmost cell of a strip s. The *length* of the strip s is $R(s) - L(s) + 1$. Each *hv*-convex 8-connected set can be considered as a sequence of strips $(s_1, \ldots, s_k)$, $k \geq 1$, where s_1 is the topmost strip and s_k is the downmost strip of the set.

Definition 1 *Let $S = (s_1, \ldots, s_k)$ be an hv-convex 8-connected set, s_l the downmost strip with $L(s_l)$ minimal and s_r the downmost strip with $R(s_r)$ maximal. We partition the set $(s_1, \ldots, s_k)$ as follows (see Fig. 3):*

- *the top* $(s_1, \ldots, s_{min(l,r)})$,
- *the interior* $(s_{min(l,r)} + 1, \ldots, s_{max(l,r)})$,
- *the bottom* $(s_{max(l,r)} + 1, \ldots, s_k)$.

We say the interior is of eastern type if $l < r$ and of western type if $l > r$.

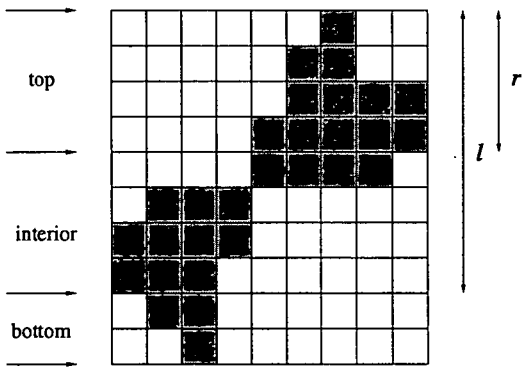


Figure 3: An *hv*-convex 8-connected discrete set with non-empty bottom. In this case $l = 8$ and $r = 4$.

The following four different types of *hv*-convex sets can arise (see Fig. 4):

- *type t* : *hv*-convex 8-connected sets with empty interior and empty bottom,
- *type* i^w: *hv*-convex 8-connected sets with empty bottom and western interior,
- *type* i^e: *hv*-convex 8-connected sets with empty bottom and eastern interior,
- *type b* : *hv*-convex 8-connected sets with non-empty bottom.

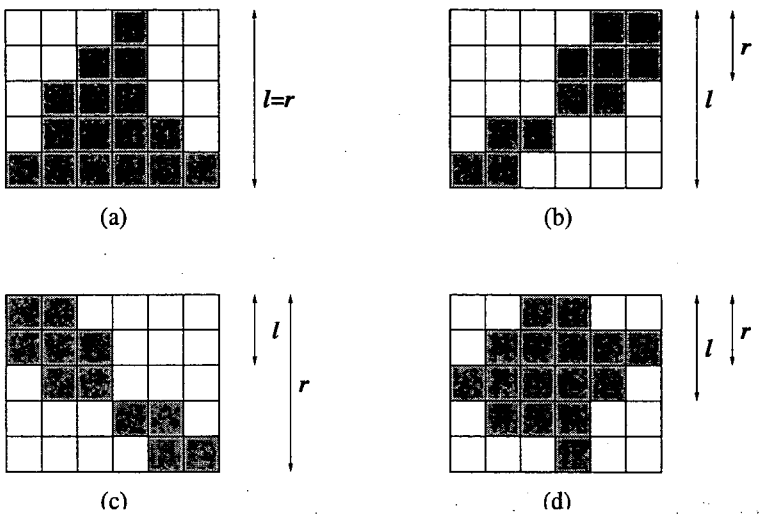


Figure 4: Four different types of *hv*-convex 8-connected sets: (a) type t; (b) type i^w; (c) type i^e ; (d) type b .

3.2 The Number of *hv*-convex 8-connected Discrete Sets

In order to count the *hv*-convex 8-connected discrete sets with fixed perimeter we construct them strip by strip starting from the top. Given a partially constructed set, we calculate its all possible extensions of this set to an *hv*-convex 8-connected discrete set.

Lemma 1 *Let $\tilde{S}(s_1, \ldots, s_j)$ be an hv-convex 8-connected set. Then the set of all extensions $(s_{j+1}, \ldots, s_k)$ of $\tilde{S}$ to an hv-convex set $S = (s_1, \ldots, s_k)$ is determined by the type of $\tilde{S}$ and the last strip s_j.*

Proof. It is similar to the proof of Lemma 1 in [14].

The number of all possible extensions of an *hv*-convex 8-connected set $\tilde{S}$ of perimeter $\tilde{n}$ to an *hv*-convex 8-connected set S of perimeter n depends on

- the type of $\tilde{S}$,
- $\tilde{m}$, the length of the downmost strip, and
- the remaining length $\tilde{l} = n - \tilde{n} + \tilde{m}$.

The cases i^e and i^w are symmetric so let denote by $N_i(\tilde{m}, \tilde{l})$ the number of possible extensions of *hv*-convex 8-connected discrete sets of interior type. Let

$N_b(\tilde{m},\tilde{l})$ and $N_t(\tilde{m},\tilde{l})$ be the number of possible extensions in the remaining two cases.

If $\tilde{l}=\tilde{m}$ then the only extension is the empty set. If $\tilde{l}<\tilde{m}$ then we cannot extend $\tilde{S}$ to an *hv*-convex 8-connected discrete sets S of perimeter n. This means, that

$$N_b(\tilde{m},\tilde{l})=N_i(\tilde{m},\tilde{l})=N_t(\tilde{m},\tilde{l})=\begin{cases}1, & \text{if } \tilde{l}=\tilde{m},\\ 0, & \text{if } \tilde{l}<\tilde{m}.\end{cases}$$

Now we count the number of extensions of $\tilde{S}$ of different types with bottom length $\tilde{m}$ and perimeter $\tilde{n}$ to *hv*-convex 8-connected sets with perimeter n ($n=\tilde{n}-\tilde{m}+\tilde{l}$ for a given remaining length $\tilde{l}$). Note that $\tilde{l}+\tilde{m}$ is always even.

- *hv*-convex 8-connected sets of type b (see Fig 5):

$$N_b(\tilde{m},\tilde{l})=\sum_{m=1}^{\tilde{m}}(\tilde{m}-m+1)N_b(m,\tilde{l}-2-\tilde{m}+m) \quad for\ \tilde{l}>\tilde{m},\ \tilde{l}+\tilde{m}\ even.$$

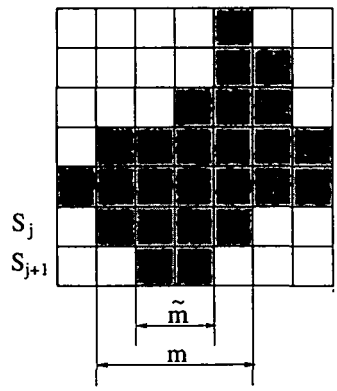


Figure 5: A possible extension of an *hv*-convex 8-connected set of type b to an *hv*-convex 8-connected set of type b.

- *hv*-convex 8-connected sets of type i (see Fig 6):

$$N_i(\tilde{m},\tilde{l})=N_i^b(\tilde{m},\tilde{l})+N_i^i(\tilde{m},\tilde{l})$$

where

$$N_i^b(\tilde{m},\tilde{l}) = \sum_{m=1}^{\tilde{m}-1}(\tilde{m}-m)N_b(m,\tilde{l}-2-\tilde{m}+m),$$

$$N_i^i(\tilde{m},\tilde{l}) = \sum_{a=0}^{\tilde{m}}\sum_{m=\tilde{m}-a}^{\frac{(\tilde{l}+\tilde{m})}{2}-a-1} N_i(m,\tilde{l}+\tilde{m}-2a-2-m).$$

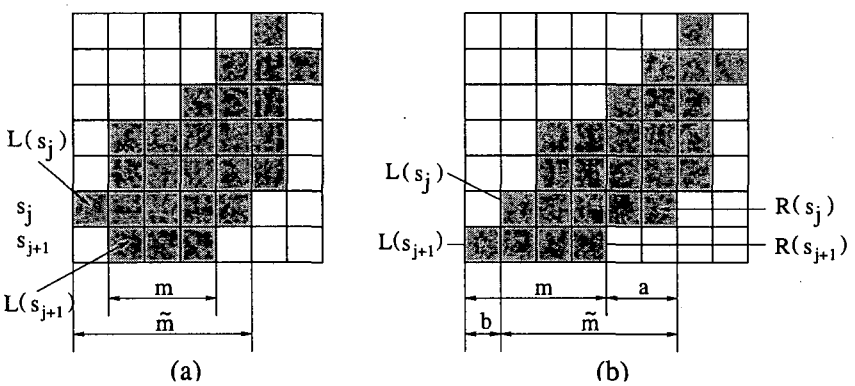


Figure 6: A possible extension of an *hv*-convex 8-connected set of type i^w to type b (a) and to type i^w (b). Notation: $a = R(s_j) - R(s_{j+1})$, $b = L(s_j) - L(s_{j+1})$.

- *hv*-convex 8-connected sets of type t (see Fig 7):

$$N_t(\tilde{m},\tilde{l}) = N_t^b(\tilde{m},\tilde{l}) + N_t^i(\tilde{m},\tilde{l}) + N_t^t(\tilde{m},\tilde{l})$$

where

$$N_t^b(\tilde{m},\tilde{l}) = \sum_{m=1}^{\tilde{m}-2}(\tilde{m}-m-1)N_b(m,\tilde{l}-2-\tilde{m}+m),$$

$$N_t^i(\tilde{m},\tilde{l}) = 2\sum_{a=1}^{\tilde{m}}\sum_{m=\tilde{m}-a}^{\frac{\tilde{l}+\tilde{m}}{2}-a-1} N_i(m,\tilde{l}+\tilde{m}-2a-2-m),$$

$$N_t^t(\tilde{m},\tilde{l}) = \sum_{m=\tilde{m}}^{\frac{\tilde{l}+\tilde{m}}{2}-1} N_t(m,\tilde{l}-2-\tilde{m}+m).$$

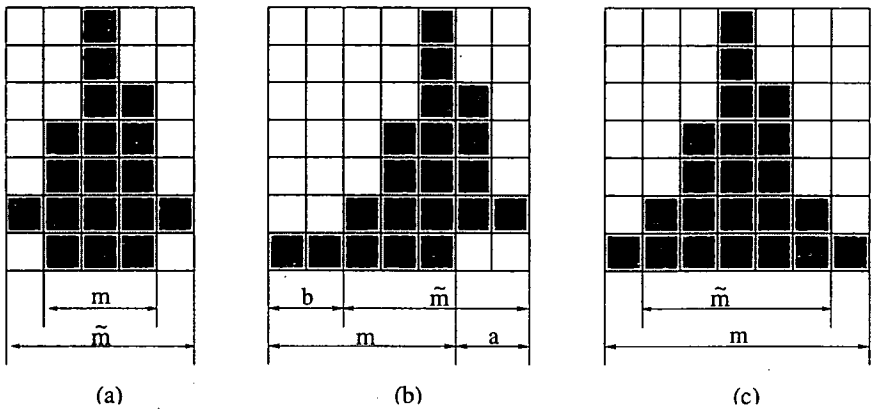


Figure 7: A possible extension of an *hv*-convex 8-connected set of type t to type b (a), to type i^w (b), and to type t (c).

Theorem 2 *The total number of hv-convex 8-connected sets of perimeter n is*

$$S_n = \sum_{\tilde{m}+2+\tilde{l}=n} N_t(\tilde{m}, \tilde{l}).$$

Remark 1 *A discrete set of type t is always 4-connected (see Fig 4.a). Extending an hv-convex 4-connected discrete set of type i into type i the result is 4-connected, if* $a = R(s_j) - R(s_{j+1}) < \tilde{m}$ *and it is 8-connected, if* $a = \tilde{m}$ *(see Fig 6.b). Extending an hv-convex 4-connected discrete set of type t into type i the result is 4-connected, if* $a < \tilde{m}$ *and it is 8-connected, if* $a = \tilde{m}$ *(see Fig 7.b). In any other cases the connectedness of the set determines the connectedness of the extended set.*

3.3 An Algorithm for Generating *hv*-convex 8-connected Discrete Sets

According to the presented way of constructing *hv*-convex 8-connected discrete sets we can construct a tree. The nodes are *hv*-convex 8-connected sets, an edge between nodes A and B means that the set represented by node A can be extended to the set represented by node B by the inclusion of a new last strip, the leaves are the *hv*-convex 8-connected sets of a given perimeter, interior nodes are partially constructed *hv*-convex 8-connected sets and the root is the empty set, which can also be considered as a trivial *hv*-convex 8-connected discrete set. In order to construct an embedding of such a tree into a finite interval of natural numbers we can use the same technique described in [14] and so we obtain the bijection between the set of *hv*-convex 8-connected sets of perimeter n and the interval $[1, S_n]$.

We fix an ordering of the summands in the presented equations which gives a

numbering of the sets. This means that for any node the set of numbers of the leaves in its subtree is an interval.

To construct a bijection between the set of *hv*-convex 8-connected sets of perimeter n and the interval $[1, P_n]$ we have to do the following:

1. Using the formulas in Subsection 3.2 compute S_n, the number of *hv*-convex 8-connected sets of perimeter n, $N_b(\tilde{m}, \tilde{l})$, $N_i(\tilde{m}, \tilde{l})$, and $N_t(\tilde{m}, \tilde{l})$ for $\tilde{m} + \tilde{l}$ even and $\tilde{m} + \tilde{l} \leq n - 2$, which give for each interior node the number of leaves of the corresponding subtrees.

2. Given an *hv*-convex 8-connected set $s = (s_1, s_2, ..., s_k)$, compute I_1, the partition of this discrete sets of perimeter n induced by s_1. Taking into consideration the type of the partially constructed part of the set, apply this procedure recursively until reaching the leaf of the tree with interval $[j, j]$.

3. Given a number j in $[1, P_n]$, compute the partition of the interval $[1, P_n]$, fix the corresponding strip s_1 and proceed recursively.

Using this bijection we can generate *hv*-convex 8-connected sets with fixed perimeter with uniform probability:

1. Compute S_n, $N_b(\tilde{m}, \tilde{l})$, $N_i(\tilde{m}, \tilde{l})$, and $N_t(\tilde{m}, \tilde{l})$ for $\tilde{m} + \tilde{l}$ even and $\tilde{m} + \tilde{l} \leq n - 2$.

2. Compute a random number j in $[1, S_n]$.

3. Apply the above presented procedure.

Remark 2 *Instead of working with perimeters we can extend the sets taking account of their area. This yields to an algorithm for calculating the number of hv-convex 8-connected discrete sets with fixed area. Working with perimeters and areas we can calculate the number of hv-convex 8-connected discrete sets with fixed perimeter and fixed area. Then with the same method as presented in this section we construct a bijection between this set and the corresponding finite interval of natural numbers. This means that we can generate hv-convex 8-connected discrete sets with fixed area or even with fixed area and fixed perimeter at random with uniform distribution.*

4 Reconstruction of *hv*-convex 8-connected Discrete Sets from Two Orthogonal Projections

In the reconstruction problem we are given two vectors $H \in \mathbb{N}^m$ and $V \in \mathbb{N}^n$. The task is to construct an *hv*-convex 8-connected discrete set S such that $\mathcal{H}(S) = H$ and $\mathcal{V}(S) = V$. The same notation is used as in [16].

4.1 Definitions

Definition 2 *An (H, V) pair of vectors is said to be compatible if there exist positive integers m, n, and A such that*

(i) $H \in \mathbb{N}^m$ *and* $V \in \mathbb{N}^n$;

(ii) $h_i \leq n$, *for* $1 \leq i \leq m$, *and* $v_j \leq m$, *for* $1 \leq j \leq n$;

(iii) $\sum_{i=1}^{m} h_i = \sum_{j=1}^{n} v_j = A$, *i.e., the two vectors have the same total sum* A.

Definition 3 *The north foot of an hv-convex 8-connected set S denoted by P_N is the set of columns of S that have elements in the first row of the rectangle Q. The column indices of P_N determine a set of consecutive integers: $\{n_f, n_f + 1, \ldots, n_l\}$.*

Similar definition can be given for the *south foot*, P_S, taking the columns of S, $\{s_f, s_f+1, \ldots, s_l\}$, which have an element in the last row of Q (see Fig. 8). The *east* and *west feet*, P_E and P_W, respectively, and their column indices, $\{e_f, e_f+1, \ldots, e_l\}$ and $\{w_f, w_f + 1, \ldots, w_l\}$, can be defined analogously.

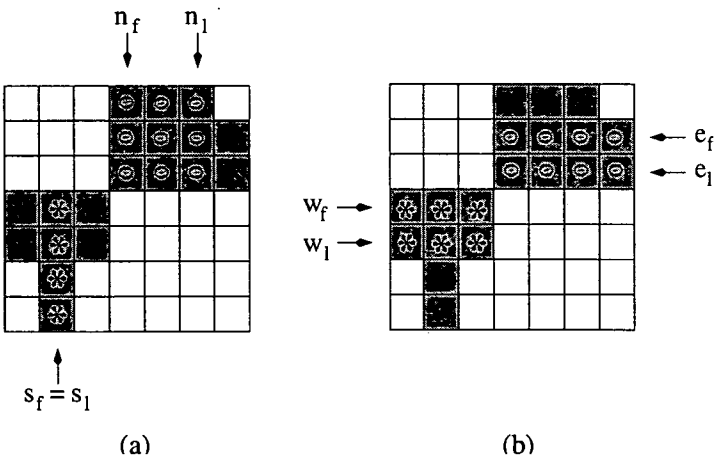


Figure 8: (a) The north and south feet of an *hv*-convex discrete set marked by o and *, respectively. (b) The east and west feet of the same discrete set marked by o and *, respectively.

Let S be an *hv*-convex 8-connected set with projections (H, V). Let $l = \max\{1 \leq j \leq n \mid v_p \leq v_q \text{ for all } 1 \leq p < q\}$, the index of the last element of the first nondecreasing subsequence of V and $r = \min\{1 \leq j \leq n \mid v_p \geq v_q \text{ for all } 1 \leq p < q \leq j\}$, the index of the first element of the last non-increasing subsequence of V. Let, furthermore, $l_1 = \min\{1 \leq j \leq l \mid v_j = v_l\}$ and,

$r_1 = \max\{r \leq j \leq n \mid v_j = v_r\}$, $l_N = \min\{l_1 + h_1 - 1, l\}$, $l_S = \min\{l_1 + h_m - 1, l\}$, $r_N = \max\{r_1 - h_1 + 1, r\}$, and $r_S = \max\{r_1 - h_m + 1, r\}$.

Then the following two propositions are true.

Proposition 1 [3] *If there is an hv-convex 8-connected set S that satisfies (H, V) with $v_j < m$ for all $j = 1, \ldots, n$, then*

1. *if P_N is to the left of P_S then $n_l \leq l_N$ and $s_f \geq r_S$,*
2. *if P_N is to the right of P_S then $s_l \leq l_S$ and $n_f \geq r_N$.*

Proof. See [3].

Proposition 2 [3] *If there is an hv-convex 8-connected set S that satisfies (H, V) with $v_j = m$ for some $j = 1, \ldots, n$ then the following four cases are possible.*

1. *If $h_1 > l - r + 1$ and $h_m > l - r + 1$ then (i) $n_f = l - h_1 + 1$, $n_l = l$, $s_f = r$, and $s_l = r + h_m - 1$, or (ii) $n_f = r$, $n_l = r + h_1 - 1$, $s_f = l - h_m + 1$, and $s_l = l$.*
2. *If $h_1 = l - r + 1$ and $h_m > l - r + 1$ then $n_f = r$, $n_l = l$, $s_f \geq \max\{1, l - h_m + 1\}$, and $s_l \leq \min\{r + h_m - 1, n\}$.*
3. *If $h_1 > l - r + 1$ and $h_m = l - r + 1$ then $n_f \geq \max\{1, l - h_1 + 1\}$, $n_l \leq \min\{r + h_1 - 1, n\}$, $s_f = r$, and $s_l = l$.*
4. *If $h_1 = h_m = l - r + 1$ then $n_f = s_f = r$, $n_l = s_l = l$.*

Proof. See [3].

Certain elements in the middle of an *hv*-convex 8-connected set can be recognized easily from the row and column sums by the following

Definition 4 *The spine of an hv-convex 8-connected set with row and column sums (H, V) is the set of positions (i, j) in T where one of the following conditions are satisfied:*

- $(n_f \leq j \leq s_l$ or $w_f \leq i \leq e_l)$ and $\widehat{v}_j \geq \widehat{h}_{i-1}$, $\widehat{h}_i \geq \widehat{v}_{j-1}$,
- $(s_f \leq j \leq n_l$ or $e_f \leq i \leq w_l)$ and $\widehat{h}_i \geq T - \widehat{v}_j$, $T - \widehat{v}_{j-1} \geq \widehat{h}_{i-1}$.

The spine of S will be denoted by S_p.

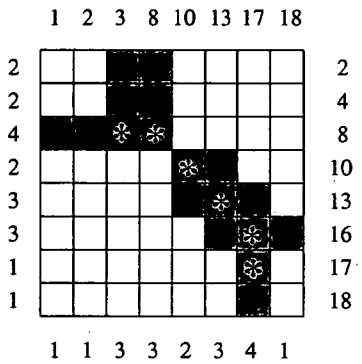


Figure 9: The spine of an *hv*-convex 8-connected discrete set marked by stars.

Definition 5 *Let S be an hv-convex 8-connected set. We say that the discrete set A is an upper-left corner region in the discrete rectangle Q containing S if $(i+1,j) \in A$ or $(i,j+1) \in A$ implies $(i,j) \in A$.*

The *upper-right*, *lower-left* and *lower-right regions*, B, C, and D, respectively, can be defined analogously (see Fig 10). Let $\overline{S}$ denote the complement of S (in Q).

Lemma 3 [5] *S is an hv-convex 8-connected discrete set if and only if $\overline{S} = A \cup B \cup C \cup D$, where A, B, C, and D are disjoint corner regions* (upper-left, upper-right, lower-left and lower-right, respectively).

Proof. See [5].

Remark 3 *$a_{i,j} = 1$ if $(i,j) \in A$, and $a_{i,j} = 0$ otherwise, for every $i = 1,...,m$ and $j = 1,...,n$. The elements of corner regions B, C, and D have a similar meaning.*

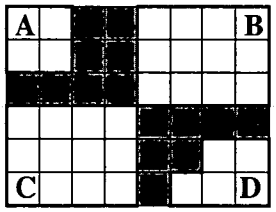


Figure 10: An *hv*-convex 8-connected discrete set with the four corner regions.

4.2 The Reconstruction Algorithm

The main idea of this algorithm is to rewrite the whole reconstruction problem as a 2-Satisfiability (2SAT) problem which can be solved in polynomial time. A 2SAT expression is a boolean expression in conjunctive normal form with at most two literals in each clause. The 2SAT problem is the following: given a 2SAT expression is there a value (true or false) for each literal that makes the expression true?

First we determine the limitations of the possible positions of the feet. Propositions 1 and 2 can be used to determine two intervals containing the possible column indices of the north and the south feet. Analogous propositions are used to determine the two intervals for the possible row indices of the east and west feet [3]. Choosing two opposite feet we can determine the spine. Without loss of generality let us suppose that we choose the west and east feet. According to the definition of the feet and the spine in this case we have found already at least one element in each column. Let $P_W = \{w_f, w_{f+1}, ..., w_l\}$ and $P_E = \{e_f, e_{f+1}, ..., e_l\}$.

Then we construct a 2SAT expression $F_{w_f,e_f}(H,V)$ such that $F_{w_f,e_f}(H,V)$ is satisfiable if and only if there is an *hv*-convex 8-connected discrete set S whose west and east feet are P_W and P_E. Let construct $F_{w_f,e_f}(H,V)$ in the following way:

$$F_{k,l}(H,V) = Cor \wedge Dis \wedge FSp \wedge LBC' \wedge UBR',$$

where Cor, Dis, FSp, LBC', and UBR' are sets of clauses describing the properties of "Corners", "Disjointness", "Feet and Spine", "Lower Bound on Column sums", and "Upper Bound on Row sums", respectively, in the following way.

$$\begin{aligned}
Cor &= \bigwedge_{i,j}(a_{ij} \Rightarrow a_{i-1,j} \ \wedge \ a_{ij} \Rightarrow a_{i,j-1}) \wedge \\
&\quad \bigwedge_{i,j}(b_{ij} \Rightarrow b_{i-1,j} \ \wedge \ b_{ij} \Rightarrow b_{i,j+1}) \wedge \\
&\quad \bigwedge_{i,j}(c_{ij} \Rightarrow c_{i+1,j} \ \wedge \ c_{ij} \Rightarrow c_{i,j-1}) \wedge \\
&\quad \bigwedge_{i,j}(d_{ij} \Rightarrow d_{i+1,j} \ \wedge \ d_{ij} \Rightarrow d_{i,j+1}), \\
Dis &= \bigwedge_{i,j}\{x_{i,j} \Rightarrow \overline{y_{i,j}} \mid \text{for symbols } X,Y \in \{A,B,C,D\},\ X \neq Y\}, \\
FSp &= \bigwedge_{S(i,j)=1}(\overline{a_{i,j}} \wedge \overline{b_{i,j}} \wedge \overline{c_{i,j}} \wedge \overline{d_{i,j}}),
\end{aligned}$$

$$LBC' = \bigwedge_{i,j}(a_{ij} \Rightarrow \overline{c_{i+v_j,j}} \wedge a_{ij} \Rightarrow \overline{d_{i+v_j,j}}) \wedge \bigwedge_{i,j}(b_{ij} \Rightarrow \overline{c_{i+v_j,j}} \wedge b_{ij} \Rightarrow \overline{d_{i+v_j,j}}),$$

$$UBR' = \bigwedge_j \left(\bigwedge_{1 \le i \le \min\{w_f, e_f\}} (\overline{a_{i,j}} \Rightarrow b_{i,j+h_i}) \wedge \bigwedge_{e_f \le i \le w_l} (\overline{a_{i,j}} \Rightarrow d_{i,j+h_i}) \right) \wedge \bigwedge_j \left(\bigwedge_{w_f \le i \le e_l} (\overline{c_{i,j}} \Rightarrow b_{i,j+h_i}) \wedge \bigwedge_{\max\{w_l, e_l\} \le i \le m} (\overline{c_{i,j}} \Rightarrow d_{i,j+h_i}) \right).$$

Then the reconstruction algorithm can be given as

Algorithm 1 *Reconstructing hv-convex 8-connected sets*

Input: Two compatible vectors $H \in \mathbb{N}^n$ and $V \in \mathbb{N}^m$.

1. Compute the cumulated sums of $\widehat{h}_i$ and $\widehat{v}_j$ for $i = 1, ..., m$ and $j = 1, ..., n$.
2. Compute the feet limitations.
3. For all possible configuration of P_W and P_E
   - 3.1. Compute the spine.
   - 3.2. If $F_{w_f,e_f}(H,V)$ is satisfiable then **Output** $S = \overline{A \cup B \cup C \cup D}$ and **halt**.

Output: Print "No solution".

Theorem 4 *$F_{w_f,e_f}(H,V)$ is satisfiable if and only if there is an hv-convex 8-connected discrete set F having projections H and V and west and east feet $P_W = \{w_f, w_{f+1}, ..., w_l\}$ and $P_E = \{e_f, e_{f+1}, ..., e_l\}$, respectively.*

Proof. Using the concepts of feet and spine, the proof is similar to the proof of Theorem 2 in [5].

In this algorithm we reduced the number of clauses in the 2SAT expression by introducing some apriori knowledge. The spine of an *hv*-convex 8-connected set guarantee the connectedness of the reconstructed object. We can define the possible feet positions from the row and column projections which reduce the number of iterations in the algorithm. The spine and two opposite feet are described by the clauses notated by FSp. The west and east feet together with the spine determine at least one cell in each column belonging to the object. This allows to reduce the number of clauses in LBC'. Using the concept of the feet we redefined the clauses which describe the upper bound on row sums (UBR').

The number of possible feet positions is smaller than m^2n^2. The complexity of the algorithm for constructing the spine for a given feet position is $O(mn)$, the 2SAT expression can be solved in $O(mn)$ time. Therefore the complexity of the algorithm for reconstruction of *hv*-convex 8-connected sets in the worst case is $O(mn \cdot \min\{m^2, n^2\})$.

Acknowledgements

The author would like to thank Attila Kuba for his valuable suggestions and remarks. This work was supported by FKFP 0908/1997 Grant of the Hungarian Ministry of Education, OTKA T032241, and by joint project MTA-NSF 123 "Aspects of Discrete Tomography" (NSF DMS9612077).

References

[1] B. Aspvall, M. F. Plass, and R. E. Tarjan, A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters* **8(3)** (1979) 121–123.

[2] E. Barcucci, A. Del Lungo, M. Nivat, R. Pinzani, Reconstructing convex polyominoes from horizontal and vertical projections, *Theor. Comput. Sci.* **155** (1996) 321–347.

[3] E. Barcucci, A. Del Lungo, M. Nivat, R. Pinzani, Medians of polyominoes: A property for the reconstruction, *Int. J. Imaging Systems and Techn.* **9** (1998) 69–77.

[4] S. Brunetti, A. Del Lungo, F. Del Ristoro, A. Kuba, M. Nivat, Reconstruction of 8- and 4-connected convex discrete sets from row and column projections, submitted for publication.

[5] M. Chrobak, C. Dürr, Reconstructing hv-convex polyominoes from orthogonal projections, *Information Processing Letters* **69** (1999) 283–289.

[6] M. P. Delest, G. Viennot, Algebraic languages and polyominoes enumeration, *Theor. Comput. Sci.* **34** (1984) 169–206.

[7] A. Del Lungo, Polyominoes defined by two vectors, *Theor. Comput. Sci.* **127** (1994) 187–198.

[8] A. Del Lungo, M. Nivat, Reconstruction of Connected Sets from Two Projections, In [13] (1999) 3-34.

[9] A. Del Lungo, M. Nivat, R. Pinzani, The number of convex polyominoes reconstructible from their orthogonal projections, *Discrete Math.* **157** (1996) 65–78.

[10] R.J. Gardner, P. Gritzmann, Uniqueness and complexity in discrete tomography, In [13](1999) 85–113.

[11] S.W. Golomb. *Polyominoes* (Schribner, New York, 1965).

[12] G.T. Herman, A. Kuba (Eds.), Discrete Tomography, Special Issue. *Int. J. Imaging Systems and Techn.* **9** (1998) No. 2/3.

[13] G.T. Herman, A. Kuba (Eds.), *Discrete Tomography: Foundations, Algorithms and Applications* (Birkhäuser, Boston, 1999).

[14] W. Hochstättler, M. Loebl, C. Moll, Generating Convex Polyominoes at Random, *Discrete Mathematics* **153** (1996) 165–176.

[15] A. Kuba, The reconstruction of two-directionally connected binary patterns from their two orthogonal projections, *Comp. Vision, Graphics, and Image Proc.* **27** (1984) 249–265.

[16] A. Kuba, Reconstruction in different classes of 2d discrete sets, In *Lecture Notes on Computer Sciences* **1586** (1999) 153-163.

[17] A. Kuba, E. Balogh, Reconstruction of convex 2D discrete sets in polynomial time, *Theor. Comput. Sci.*, to appear.

[18] H.J. Ryser, Combinatorial properties of matrices of zeros and ones, *Canad. J. Math.* **9** (1957) 371–377.

[19] G.W. Woeginger, The reconstruction of polyominoes from their orthogonal projections, *Techn. Report, Technische Universität Graz* **65** (1996).

Acta Cybernetica **15** (2001) 201–211.

A 3D Parallel Shrinking Algorithm

Kálmán Palágyi*

Abstract

Shrinking is a frequently used preprocessing step in image processing. This paper presents an efficient 3D parallel shrinking algorithm for transforming a binary object into its topological kernel. The applied strategy is called directional: each iteration step is composed of six subiterations each of which can be executed in parallel. The algorithm makes easy implementation possible, since deletable points are given by $3 \times 3 \times 3$ matching templates. The topological correctness of the algorithm is proved for $(26, 6)$ binary pictures.

1 Introduction

A 3D binary picture [6] is a mapping that assigns the value of 0 or 1 to each point with integer coordinates in the 3D digital space denoted by $\mathbb{Z}^3$. Points having the value of 1 are called black points and form the objects in the picture, while 0's are called white points and form the background, the cavities, and the holes in the picture.

Shrinking of binary pictures into similarly connected representations that have smaller foregrounds (i.e., fewer 1's) has found application as a fundamental preprocessing step in image processing [3]. Two forms of such shrinking have been emerged:

1. The picture is transformed into its topological kernel, where the shrunk picture is topologically equivalent to the original one.
2. Each object (connected components of 1's) in the picture is shrunk into an isolated point (i.e., single–point residue which may then be deleted). Obviously, this kind of shrinking alters the topology of the original picture, since holes and cavities are eliminated.

As far as we know, the only 3D topology preserving shrinking algorithm has been proposed by Bertrand and Aktouf [2]. Their thinning algorithm can extract the topological kernel of an object if no end–point condition is applied. The strategy which is used for deleting 1's in parallel without altering the topology of the picture is based upon subfields: the cubic grid $\mathbb{Z}^3$ is divided into 8 subfields which are

*Department of Applied Informatics, University of Szeged, H-6701 Szeged P.O.Box 652, Hungary, E–mail: palagyi@inf.u-szeged.hu

successively activated in each iteration step. The parallel algorithm examines the $3 \times 3 \times 3$ neighbourhood of the object points.

Two 3D shrinking algorithms belonging to the second type are known: Arcelli and Levialdi [1] proposed a parallel algorithm capable of transforming any finite object to an isolated 1 in a finite number of iteration step. Only the $2 \times 2 \times 2$ neighbourhood of 1's are investigated and the object to be shrunk never leaves its circumscribing box. Hall and Küçük [4] developed the other algorithm which uses 2 subfields and examines the $3 \times 3 \times 3$ neighbourhood of the object points.

In this work, a new 3D parallel shrinking algorithm is proposed for extracting the topological kernel of a binary picture. Our strategy used to preserve the topology is called directional or border sequential: Iteration steps are divided into 6 successive parallel subiterations, where only border 1's of a certain kind can be deleted in each subiteration. The algorithm examines the $3 \times 3 \times 3$ neighbourhood of 1's and it is topology preserving for any $(26, 6)$ pictures.

2 Basic Notions and Results

Let p be a point in the 3D digital space $\mathbb{Z}^3$. Let us denote $N_j(p)$ (for $j = 6, 18, 26$) the set of points *j-adjacent* to a point p (see Fig. 1).

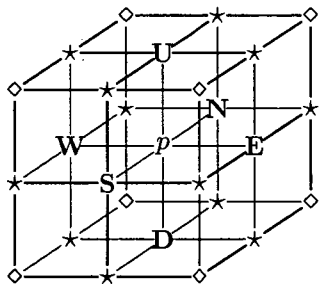


Figure 1: Frequently used adjacencies in $\mathbb{Z}^3$. The set $N_6(p)$ contains the central point p and points marked **U**, **N**, **E**, **S**, **W**, and **D**. The set of points $N_{18}(p)$ contains the set $N_6(p)$ and points marked "$\star$". The set of points $N_{26}(p)$ contains the set $N_{18}(p)$ and points marked "$\diamond$".

The sequence of distinct points $\langle x_0, x_1, \ldots, x_n \rangle$ is a *j-path* of length n from point x_0 to point x_n in a non-empty set of points X if each point of the sequence belongs to X and x_i is j-adjacent to x_{i-1} for each $1 \leq i \leq n$. Note that a single point is a j-path of length 0. Two points are *j-connected* in the set X if there is a j-path in X between them. A set of points X is *j-connected* in the set of points $Y \supseteq X$ if any two points in X are j-connected in Y.

The *3D binary (m,n) digital picture* $\mathcal{P}$ is a quadruple $\mathcal{P} = (\mathbb{Z}^3, m, n, B)$ [6]. Each element of $\mathbb{Z}^3$ is called a *point* of $\mathcal{P}$. Each point in $B \subseteq \mathbb{Z}^3$ is called a *black point* and value 1 is assigned to it. Each point in $\mathbb{Z}^3 \backslash B$ is called a *white point* and value 0 is assigned to it. Adjacency m corresponds to the black points and

adjacency n corresponds to the white points. A *black component* or an *object* is a maximal m–connected set of points in B. A singleton black component is called *isolated point*. A *white component* is a maximal n–connected set of points in $\mathbb{Z}^3 \backslash B$.

We are dealing with (26,6) pictures. It is assumed that any picture contains finitely many black points. In a finite picture, there is a unique infinite white component, which is called the *background*. A finite white component (surrounded by an object) is called a *cavity*.

A black point p is called a *border point* if the set $N_6(p)$ contains at least one white point. A border point p is called a **U**–*border point* if the point marked by **U** in Fig. 1 is white. We can define **N**–, **E**–, **S**–, **W**–, and **D**–border points in the same way.

Our shrinking algorithm can be regarded as a reduction operation that changes some black points to white ones but does not alter white points.

A reduction operation does *not* preserve topology if

- any object in the input picture is split (into two or more parts) or completely deleted (i.e., each point belonging to the object is deleted),
- any cavity in the input picture is merged with the background or with another cavity, or
- a cavity is created where there was no cavity in the input picture.

There is an additional concept called *hole* or *tunnel* in 3D pictures. A hole (that a doughnut has) is formed by white points, but it is not a cavity [6]. Topology preservation implies that eliminating or creating holes is not allowed.

A black point is called a *simple point* if its deletion does not alter the topology of the picture [10]. We make use of the following result for (26,6) pictures:

Criterion 1. [9, 13]
Black point p is simple in picture $(\mathbb{Z}^3, 26, 6, B)$ if and only if all of the following conditions hold:

1. *the set $(B \backslash \{p\}) \cap N_{26}(p)$ contains exactly one 26–component; and*
2. *the set $(\mathbb{Z}^3 \backslash B) \cap N_6(p)$ is not empty and it is 6–connected in the set $(\mathbb{Z}^3 \backslash B) \cap N_{18}(p)$.*

The *topological kernel* of an object is topologically equivalent to the original object and there is no simple point in it. (In other words, the topological kernel can be get by the sequential deletion of simple points.)

Parallel reduction operations delete a set of black points and not only a single simple point (and each white point remains the same). We need to consider what is meant by topology preservation when a number of black points are deleted simultaneously. Some sufficient (but not necessary) conditions have been stated for parallel reduction operations [5, 7, 8]. We make use of the following result for (26,6) pictures:

Theorem 2. [11, 12]
Let $\mathcal{T}$ be a parallel reduction operation. Let p be any black point in any picture

$\mathcal{P} = (\mathbb{Z}^3, 26, 6, B)$ so that p is deleted by $\mathcal{T}$. Let $Q \subseteq (N_{18}(p) \backslash \{p\}) \cap B$ be any set of black points in picture $\mathcal{P}$. Operation $\mathcal{T}$ is topology preserving for (26,6) pictures if all of the following conditions hold:

1. *p is simple in the picture $(\mathbb{Z}^3, 26, 6, B \backslash Q)$; and*
2. *no black component contained entirely in a unit lattice cube (i.e., a $2 \times 2 \times 2$ configuration in $\mathbb{Z}^3$) can be deleted completely by operation $\mathcal{T}$ (i.e., at least one point in such an object must be preserved).*

3 The New Shrinking Algorithm

The proposed directional 6–subiteration shrinking algorithm can be sketched by the following program:

Input: picture $\mathcal{P} = (\mathbb{Z}^3, 26, 6, B)$;
Output: picture $\mathcal{P}' = (\mathbb{Z}^3, 26, 6, B')$.

```
6_subiteration_shrinking_algorithm(B,B')
   begin
      B' = B;
      repeat
         B' = deletion_from_U(B');
         B' = deletion_from_D(B');
         B' = deletion_from_N(B');
         B' = deletion_from_S(B');
         B' = deletion_from_E(B');
         B' = deletion_from_W(B');
      until no points are deleted;
   end.
```

The first subiteration (`deletion_from_U`) corresponding to the deletion direction **U** can delete certain **U**–border points; the second subiteration associated with the deletion direction **D** attempts to delete **D**–border points, and so on.

Our algorithm terminates when there are no more black points to be deleted. Since all considered input pictures are finite, the shrinking algorithm will terminate.

Deletable points in a subiteration are given by a set of $3 \times 3 \times 3$ matching templates. Templates are described by three kinds of elements, "black", "white", and "don't care". Each template defines a predicate: a black point p satisfies the predicate if the given template matches $N_{26}(p)$, where each black template element coincides with a black point, each white template element coincides with a white point, and a "don't care" element in the template coincides with either a white point or a black point. Note that no reflection or rotation is allowed in matching a template to $N_{26}(p)$. A black point is deletable if at least one template in the given set of templates matches it.

The set of templates assigned to the first subiteration is given by Fig. 2. Note that Fig. 2 shows only the five base templates **T1**–**T5**. Additionally, all their

rotations around the vertical axis belong to $\mathcal{T}_{\mathbf{U}}$, where the rotation angles are 90°, 180°, and 270°. The deletable points of the other five subiterations can be obtained by proper rotations and/or reflections of the templates in $\mathcal{T}_{\mathbf{U}}$.

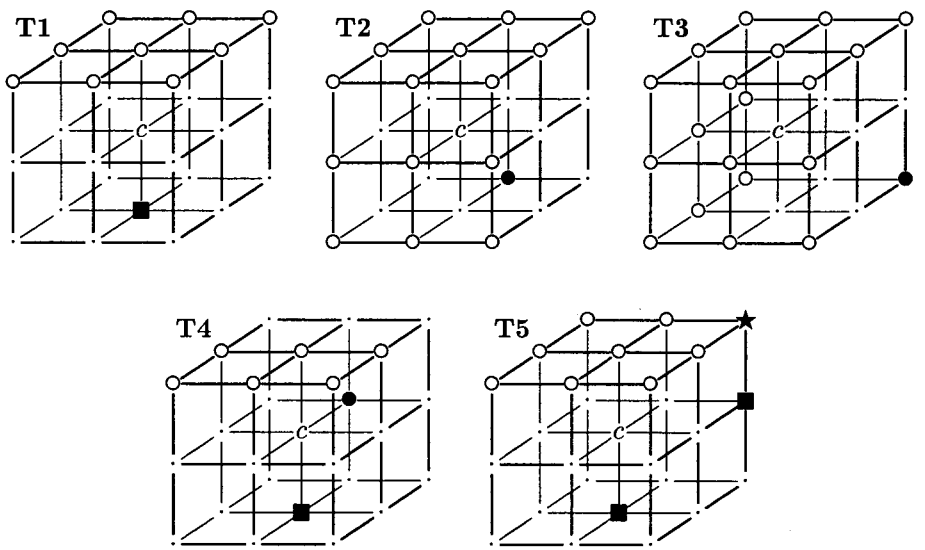


Figure 2: Base templates **T1**–**T5** and their rotations around the vertical axis form the set of templates $\mathcal{T}_{\mathbf{U}}$ assigned to the first subiteration of the proposed algorithm. These templates can delete certain **U**–border points. Notations: each position marked "*c*", "●", "■", and "★" matches a black point; each position marked "○" matches a white point; every "·" ("don't care") matches either a black or a white point. (Note that using four different symbols for black template positions helps us to prove the topological correctness of the algorithm.)

Note that choosing another order of the deletion directions yields another algorithm, but it does not alter the topological correctness.

The templates of our algorithm can be regarded as a "careful" characterization of simple points. This Boolean characterization makes easy implementation possible.

4 Discussion

The proposed algorithm has been tested on objects of different shapes. Here we present only three examples (see Fig. 3).

An object is *simply-connected* if it has no holes nor cavities [6]. (In other words, a simply-connected object is topologically equivalent to a solid sphere.) The proposed algorithm is capable of transforming each simply-connected object to an isolated point. Each *multiply-connected object* (i.e., object having either holes

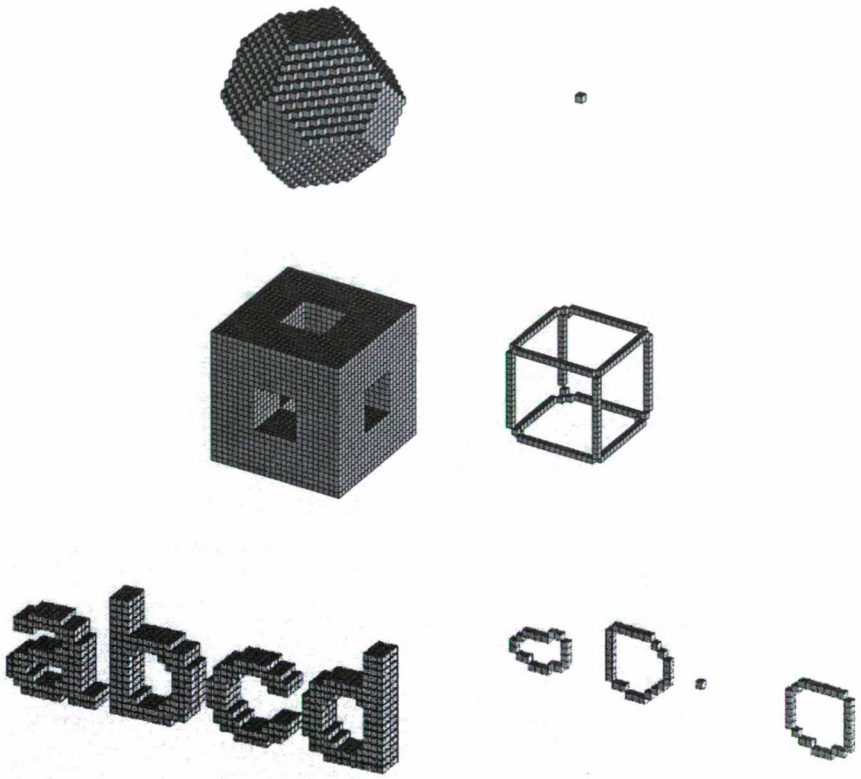


Figure 3: Three synthetic pictures (left) and their topological kernels produced by the proposed shrinking algorithm (right).

or cavities; e.g., a coffee cup or a cheese) is transformed into a closed "thin" curve or into a closed "thin" surface.

The proposed 6–subiteration shrinking algorithm is topology preserving for $(26,6)$ pictures. It is sufficient to prove that reduction operation given by the set of templates $\mathcal{T}_{\mathrm{U}}$ is topology preserving. If the first subiteration of the algorithm is topology preserving, then the others are topology preserving, too, since the reflections and rotations of the deletion templates do not alter their topological properties. Therefore, the entire algorithm is topology preserving, since it is composed of topology preserving reductions.

In order to prove both conditions of Theorem 2, we classify the elements of templates and state some properties of the set of templates $\mathcal{T}_{\mathrm{U}}$. The element in the very centre of a template is called *central* (marked by "c" in Fig. 2). A noncentral template element is called *black* if it is always black (marked by "●", "■", and "★" in Fig. 2). A noncentral template element is called *white* if it is always white

(marked by "o" in Fig. 2). Any other noncentral template element which is not white and not black, is called *potentially black* (marked by "·" in Fig. 2). A black or a potentially black non–central template element is called *nonwhite*. A black point p is *deletable* if it can be deleted by at least one template in $\mathcal{T}_U$; p is *nondeletable* otherwise.

Observation 3. *Let us examine the configurations illustrated in Fig. 4.*

1. *If black point p in the configuration (a) is deletable then point q is white.*
2. *If both points p and q in the configuration (a) are black and point q is deletable then point p is nondeletable.*
3. *If black point p in the configuration (b) is deletable then at least one point marked q is black.*

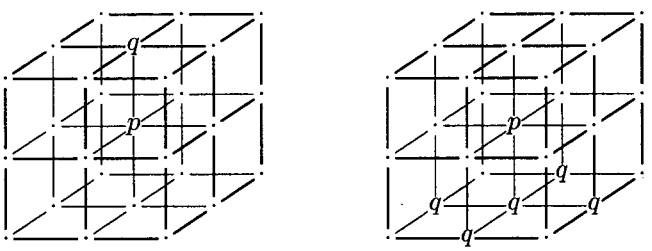


Figure 4: Configurations assigned to Observation 3.

The topological correctness of the first subiteration of the proposed algorithm is stated by the following theorem:

Theorem 4.
Reduction operation given by the set of templates $\mathcal{T}_U$ is topology preserving for (26, 6) *pictures.*

Proof.
It is to be proved that both conditions of Theorem 2 are satisfied.

It is easy to see that each template in $\mathcal{T}_U$ deletes only simple points of (26,6) pictures.

The first point is to verify that there exists a 26–path between any two nonwhite positions (condition 1 of Criterion 1). It is sufficient to show that any potentially black position is 26–adjacent to a black position and any black position is 26–adjacent to another black position. It is obvious by careful examination of the templates in $\mathcal{T}_U$.

To prove that condition 2 of Criterion 1 holds, it is sufficient to show for each template in $\mathcal{T}_U$ that:

1. there exists a white position 6–adjacent to the central position,

2. for any two white positions 6–adjacent to the central position p are 6–connected in the set of white positions 18–adjacent to p,
3. and for any potentially black position 6–adjacent to the central position p, there exists a 6–adjacent white 18–neighbour which is 6–adjacent to a white position 6–adjacent to p.

The first point holds by Observation 3/1. The remaining two points are obvious by a careful examination of the templates in $\mathcal{T}_{\mathrm{U}}$.

We know that each deletable point p is simple. It can be stated that the value of any point coinciding with a potentially black template position does not alter the simplicity of p. We can state that the simplicity of a point p does not depend on the points that coincide with a template position marked "★" (see Fig. 2). In addition, black points that coincide with template positions marked "■" are nondeletable (by Observation 3/2). Therefore, it is sufficient to deal with deletable points that coincide with template positions marked "●". Note that base templates **T1** and **T5** (and their rotated versions) do not contain any positions marked "●". Therefore, only base templates **T2**, **T3**, and **T4** (and their rotated versions) are to be investigated.

Let us consider a black point p which can be deleted by template **T2**, **T3**, or **T4** (or their rotated versions) and let $Q \subseteq (N_{18}(p) \backslash \{p\}) \cap B$ be a nonempty set of deletable points. (If $Q = \emptyset$ then p remains simple after the deletion of Q, since each deletable point is simple.) Two cases are to be distinguished:

1. A black point $q \in Q$ does not coincide with the only element marked "●" of the template that may delete p.
   In this case, each point in Q must coincide with a potentially black position, therefore, the simplicity of p is not altered by the deletion of the set Q.
2. A black point $q \in Q$ coincides with the only element marked "●" of the template that may delete p.
   (a) Suppose that p is deleted by template **T2** (or one of its rotated versions) and let us consider the configurations in Fig. 5. We can state that point q may be deleted only by a rotated version of template **T4**. In this case, point r must be black and nondeletable (see 5/b), therefore, point p can be deleted by template **T1** (see 5/a) after the deletion of Q. Consequently, point p remains simple.
   (b) Suppose that p is deleted by template **T3** (or one of its rotated versions) and let us consider the configurations in Fig. 6. We can state that point q may be deleted only by a rotated version of template **T5**. In this case, point r must be black and nondeletable (see 6/b), therefore, point p can be deleted by template **T1** (see 6/a) after the deletion of Q. Consequently, point p remains simple.
   (c) Suppose that p is deleted by template **T4** (or one of its rotated versions) and let us consider the configurations in Fig. 7.

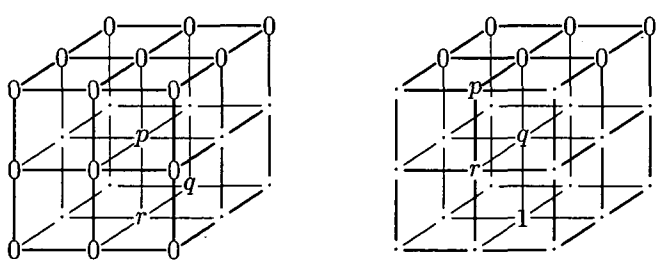
i. If point q may be deleted by one of the templates **T1**, **T2**, **T3**, **T5** (or by one of their rotated versions), or two rotated versions of template **T4** (where rotation angles are 0° and 180°) then all points r, s, and t are white (see 7/b), therefore, point p can be deleted by template **T1** (see 7/a) after the deletion of Q. Consequently, point p remains simple.

ii. If point q may be deleted by the rotated version of template **T4** (where the rotation angle is 90°) then $r = s = 0$ and $v = 1$ (see 7/b). If $t = 0$ then p can be deleted by template **T1**. If $t = 1$ then p can be deleted by template **T5** (and v is nondeletable, therefore, $v \notin Q$). Consequently, point p remains simple after the deletion of Q.

iii. If point q may be deleted by the rotated version of template **T4** (where the rotation angle is 270°) then $s = t = 0$ and $w = 1$ (see 7/b). If $r = 0$ then p can be deleted by template **T1**. If $r = 1$ then p can be deleted by a rotated version of template **T5** (and w is nondeletable, therefore, $w \notin Q$). Consequently, point p remains simple after the deletion of Q.

Therefore, condition 1 of Theorem 2 is satisfied.

Condition 2 of Theorem 2 can be seen with the help of Observation 3, too. Let us consider a unit lattice cube containing an upper set of four points $U = \{u_1, u_2, u_3, u_4\}$ and a lower set of four points $L = \{l_1, l_2, l_3, l_4\}$ (see Fig. 8). Let $C \subseteq U \cup L$ be a black component contained entirely in the unit lattice cube. If $C \cap U$ contains a deletable point then $C \cap L \neq \emptyset$ by Observation 3/3. It is easy to see, that any point in $C \cap L$ is nondeletable by Observation 3/3. Therefore, black component C cannot be deleted completely.



(a) (b)

Figure 5: The $3 \times 3 \times 3$ configuration if point p is deleted by template **T2** (a) and the only configuration if point q is deletable (b).

5 Conclusions

Shrinking is fundamental operation in image processing. For example, shrinking can be used as a preprocessing step for counting distinct objects in a binary picture or to perform object labeling. An efficient 3D parallel shrinking algorithm for reducing a binary object to its topological kernel is presented. The applied directional strategy allows isotropic erosion, therefore, the residue is in the "middle" of an object. The topological correctness of the proposed algorithm is proved.

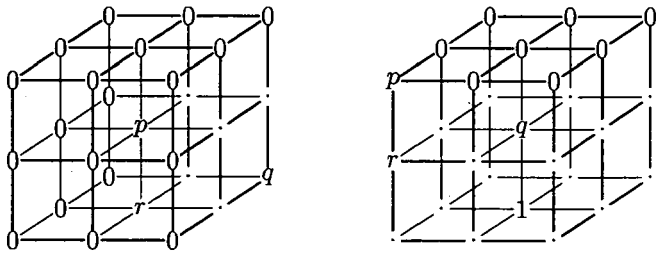


Figure 6: The $3 \times 3 \times 3$ configuration if point p is deleted by template **T3** (a) and the only configuration if point q is deletable (b).

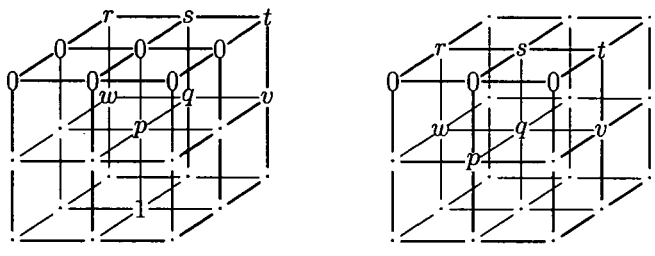


Figure 7: The $3 \times 3 \times 3$ configuration if point p is deleted by template **T4** (a) and the corresponding configuration of $N_{26}q$ (b).

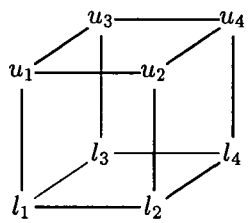


Figure 8: A unit lattice cube divided into two sets of four points $U = \{u_1, u_2, u_3, u_4\}$ and $L = \{l_1, l_2, l_3, l_4\}$.

Acknowledgment

The author is grateful to László G. Nyúl for his valuable suggestions. This work was supported by the FKFP 0908/1997 Grant.

References

[1] C. Arcelli, S. Levialdi, Parallel shrinking in three dimensions, *Computer Graphics and Image Processing* **1**, 1972, 21–30.

[2] G. Bertrand, Z. Aktouf, A 3D thinning algorithms using subfields, in *Proc, Conf. on Vision Geometry III*, Proc. SPIE **2356**, 1994, 113–124.

[3] R.W. Hall, T.Y, Kong, A. Rosenfeld, Shrinking binary images, in *Topological Algorithms for Digital Image Processing*, Machine Intelligence and Pattern Recognition 19, North–Holland, 1996, 31–98.

[4] R.W. Hall, Ş. Küçük, Parallel 3D shrinking algorithms using subfields notions, in *Proc. 11th IEEE Int. Conf. Pattern Recognition*, 1992, 395–398.

[5] T.Y. Kong, On topology preservation in 2–d and 3–d thinning, *Int. J. of Pattern Recognition and Artifical Intelligence* **9**, 1995, 813–844.

[6] T.Y. Kong, A. Rosenfeld, Digital topology: Introduction and survey, *Computer Vision, Graphics, and Image Processing* **48**, 1989, 357–393.

[7] C.M. Ma, On topology preservation in 3D thinning, *CVGIP: Image Understanding* **59**, 1994, 328–339.

[8] C.M. Ma, Connectivity preservation of 3D 6–subiteration thinning algorithms, *Graphical Models and Image Processing* **58**, 1996, 382–386.

[9] G. Malandain, G. Bertrand, Fast characterization of 3D simple points, in *IEEE Int. Conf. Pattern Recognition*, 1992, 232–235.

[10] D.G. Morgenthaler, *Three–dimensional simple points: Serial erosion, parallel thinning and skeletonization*, Technical Report TR–1005, Computer Vision Laboratory, Computer Science Center, University of Maryland, 1981.

[11] K. Palágyi, A. Kuba, A 3D 6–subiteration thinning algorithm for extracting medial lines, *Pattern Recognition Letters* **19**, 1998, 613–627.

[12] K. Palágyi, A. Kuba, A parallel 3D 12–subiteration thinning algorithm, *Graphical Models and Image Processing* **61**, 1999, 199–221.

[13] P.K. Saha, B.B. Chaudhuri, Detection of 3–D simple points for topology preserving transformations with application to thinning, *IEEE Trans. Pattern Analysis and Machine Intelligence* **16**, 1994, 1028–1032.

Acta Cybernetica **15** (2001) 213-224.

Learning Decision Trees in Continuous Space

J. Dombi* and Á. Zsiros†

Abstract

Two problems of the ID3 and C4.5 decision tree building methods will be mentioned and solutions will be suggested on them. First, in both methods a *Gain*-type criteria is used to compare the applicability of possible tests, which derives from the entropy function. We are going to propose a new measure instead of the entropy function, which comes from the measure of fuzziness using a monotone fuzzy operator. It is more natural and much simpler to compute in case of concept learning (when elements belong to only two classes: positive and negative).

Second, the well-known extension of the ID3 method for handling continuous attributes (C4.5) is based on discretization of attribute values and in it the decision space is separated with axis-parallel hyperplanes. In our proposed new method (CDT) continuous attributes are handled without discretization, and arbitrary geometric figures are used for separation of decision space, like hyperplanes in general position, spheres and ellipsoids. The power of our new method is going to be demonstrated on a few examples.

1 Introduction

A lot of applications on the area of artificial intelligence and data mining lead to a similar task: constructing a classification model based on our knowledge. Classification models help us to understand the underlying structure of a given problem and later they can be used to predict classes of unseen elements.

Classification models could be grouped by the way of construction: they are made by human experts or obtained from a set of examples, inductively. The built up model may be a non-hierarchical one (like a instance-based classifier, or a model obtained from a neural network, a genetic algorithm and a statistical method) or a hierarchical one like a decision tree (for a short overview and further references see [1]). We will deal with hierarchical classifiers built up inductively.

Throughout this article T denotes the given set of elements with their class information (training set) from which we would like to build up a decision tree:

$$T = \{(\mathbf{x}, c(\mathbf{x}))|\mathbf{x} \in X\},$$

*Department of Applied Informatics, University of Szeged, Árpád tér 2, H-6720 Szeged, Hungary, e-mail: dombi@inf.u-szeged.hu

†Department of Applied Informatics, University of Szeged, Árpád tér 2, H-6720 Szeged, Hungary, e-mail: zsiros@inf.u-szeged.hu

where $\mathbf{x}$ is an element of X, described by a sequence of attribute values: $\mathbf{x} = (a_1, \ldots, a_n)$, a_i is the value of the ith attribute, n is the number of attributes, and $c(\mathbf{x})$ gives the class of element $\mathbf{x}$. Finally denote $C_1, \ldots, C_k$ the possible classes of elements of T.

Usually two different types of attributes are distinguished: an attribute is discrete (categorical), if its value comes from a predefined finite set; and continuous (numerical), if it may be any element of a (real) interval.

Now, we are ready to give the definition of decision tree:

Definition 1 *Decision tree is a special rooted tree, in which a class identifier is associated to each leaf node (that determinates the class of elements reached that node), and each internal (or decision) node specifies some test, with one branch and subtree for each outcome of the test.*

Decision tree building methods might be grouped by the variety of tests used in their inner nodes. In some methods only single attribute selection is allowed as a test, for example in Quinlan's ID3 and in its extension to handle continuous attributes C4.5[1]. In other methods some mixture of attributes are also possible as tests, for instance in Cios's CID3[5] and in our new CDT method.

Thus, the final task is to make a decision tree from a given training set, where elements are described by some (discrete or continuous) attributes. Of course it is trivial to build up a tree consistent with the training set[4] (imagine a decision tree in which all leaf nodes contain only one element of the training set). But this tree does not tell us anything about the structure of the original problem. As Occam's razor tells us, we should look for one of the smallest decision trees. This philosopher idea says that a simple decision tree might perform better on unseen elements than a more complex one (see [10, 11]). Unfortunately, *the problem of finding the smallest decision tree consistent with a training set is NP-Complete[3]* as Hyafil and Rivest have already shown that.

To cope with this computational problem, a greedy, divide-and-conquer algorithm is used to build up a decision tree consistent with the training set (which, of course, usually leads not the smallest one). First, the one-node decision tree is taken (containing all elements of the training set). Later, in each step a test is selected and applied on the examples reached the current node. Then the test is assigned to the node and branches are made according to the outcomes of the selected test. Finally, the same method is applied on these newly created branches. It may be seen, that the critical point of this algorithm is the test selection criteria. The following methods (ID3, C4.5, CDT) differ only at this point.

In our CDT method a new function is used to measure the homogeneity of examples instead of the entropy function (which is used in ID3 and C4.5). It is similar to the entropy function in case of concept learning (which means that examples belong to only two classes). Unfortunately CDT can handle only such problems. On the other hand the decision space, described by a set of continuous attributes, is partitioned by arbitrary geometric figures in the CDT method, while in C4.5 only axis-parallel hyperplanes are used. These tests finally lead to a binary decision tree that correctly classifies all elements of the training set.

In the next section the test selection criteria of the ID3 and the C4.5 methods are introduced. Section 3 mentions some results from fuzzy theory that will be used later to introduce the new test selection criteria in section 4. Its work is demonstrated on an example. Finally the new decision tree building method is described in section 5 and its power is demonstrated on a few examples in section 6 followed by conclusions in section 7.

2 The ID3 and the C4.5 methods

In this section we are going to give a short description of the ID3 method, of its extension to handle continuous attributes (C4.5) and we are going to demonstrate their work through an example.

As it has been mentioned earlier, the crucial points of the tree-building algorithm are the variety of possible tests and the test selection criteria. In the ID3 and C4.5 methods tests are based on single attribute selection, so the possible tests are about the possible attributes.

The idea of test selection is to examine training examples and to find the attribute that separates the examples most perfectly considering their class membership (as the greedy approach suggests it). The ID3 algorithm uses a function coming from the field of information theory to measure the entropy in the original training set and in the subsets after partitioning. Formally the criteria is the following (where $|C_j|_T$ denotes the number of elements of T belong to class C_j):

$$\max_A \{Gain(A)\}$$

where

$$Gain(A) = I(T) - E(A,T)$$

$$I(T) = -\sum_{j=1}^{k} \frac{|C_j|_T}{|T|} \cdot \log_2 \left(\frac{|C_j|_T}{|T|} \right)$$

is the entropy function and

$$E(A,T) = \sum_{i=1}^{m} \frac{|T_i|}{|T|} \cdot I(T_i)$$

is the weighted sum of the entropies of subsets. It is important to remark, that the application of entropy function and the *Gain* criteria, which is just the simple difference of $I(T)$ and $E(A,T)$, has no theoretical background, it is just a heuristic!

Unfortunately, the *Gain* criteria is biased towards discrete attributes with more outcomes, thus it has a modification, the *Gain ratio* test selection criteria (see [1]) which reduces the value of *Gain* in case of many valued discrete attributes.

The ID3 method can be used only on problems described by discrete attributes. Although it has an extension to handle continuous attributes (C4.5), in this method

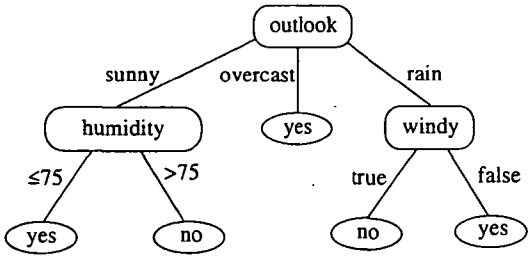


Figure 1: Decision tree built up from the training set on table 1

outlook	Temp(F)	Humidity(%)	Windy?	Class
sunny	75	70	true	Play
sunny	80	90	true	Don't Play
sunny	85	85	false	Don't Play
sunny	72	95	false	Don't Play
sunny	69	70	false	Play
overcast	72	90	true	Play
overcast	83	78	false	Play
overcast	64	65	true	Play
overcast	81	75	false	Play
rain	71	80	true	Don't Play
rain	65	70	true	Don't Play
rain	75	80	false	Play
rain	68	80	false	Play
rain	70	96	false	Play

Table 1: The training set

the attributes are discretizated, handled like discrete ones with two outcomes. Different values of the candidate continuous attribute are ordered: $v_1 \leq v_2 \leq \ldots \leq v_n$ and all $\frac{v_i+v_{i+1}}{2}$, $i = 1, \ldots, n-1$ midpoints are checked as possible thresholds to divide the training set into two partition.

The test selection criteria in C4.5 is biased towards continuous attributes with numerous distinct values. Its examination and a modified criteria are found in [2].

There is a decision tree, built up from a 12 element training set on figure 1. This example is well-known from the literature [1, 11] and shows when to play a specific game. The test selection criteria is maximization of *Gain*, so first we choose attribute 'outlook'. Then the algorithm is called recursively to the first and third branches of the tree. Finally, we get a decision tree which classifies all training elements correctly.

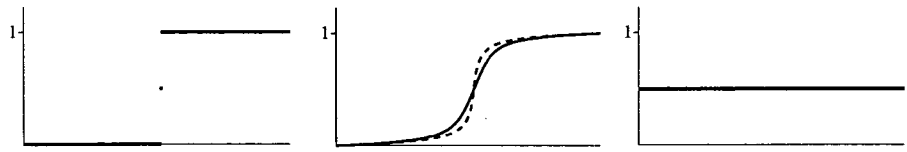


Figure 2: Membership functions with different sharpness

3 Measure of fuzziness

In this section a few basic results are mentioned about the general class of fuzzy operators and the measure of fuzziness.

There are two main classes of associative, subidempotence conjunction operators in the fuzzy theory: the class of strictly monotone and the class of nilpotens operators[6]. Monotone conjunction operators may be written in form $c(x,y) = f^{-1}(f(x) + f(y))$ where $f(x) : (0,1] \to \Re^+$, $\lim_{x\to 0} = \infty$, $f(1) = 0$ and $f(x)$ is a strictly decreasing function; and nilpotens operators in form $c(x,y) = f^{-1}([f(x) + f(y)])$ where $[x]$ is the cut function

$$[x] = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \le x \le 1 \\ 1 & \text{otherwise} \end{cases}$$

and $f(x) : [0,1] \to [0,1]$, $f(0) = 1$, $f(1) = 0$ is a strictly decreasing function. These general forms give us a lot of fuzzy conjunction operators: one for each suitable generator function $f(x)$. Disjunction operators could be characterized in the same manner with a bit different assumptions for generator function $f(x)$.

By the help of conjunction operators we are able to derive a function that measures the fuzziness of a (membership) function. For example in figure 2 some different membership functions are shown. The first one makes a very sharp switch from 0 to 1, so it is not fuzzy, on the second picture the dotted one is a less fuzzy while the other one is a more fuzzy function. On the third picture a total fuzzy function is shown. This property is measured with the following formula (in discrete case)[6]:

$$S = \frac{1}{n} \sum_{i=1}^{n} F(x_i)$$

where $c(x,y)$ is a conjunction operator (as above) and

$$F(x) = \frac{1}{c\left(\frac{1}{2}, \frac{1}{2}\right)} \cdot c(x, 1-x).$$

This function will be useful in the next section to measure how separated the elements of the training set are. It is also interesting to note that the entropy function, used in the ID3 method, may be obtained from this measure of fuzziness in a special case.

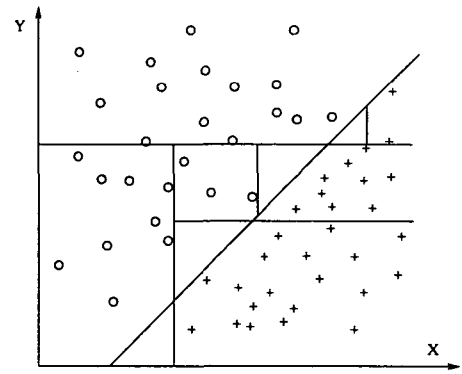


Figure 3: Geometric interpretation of classification

4 The new test selection criteria

Throughout the following sections we assume that elements of the training set belong to only two classes: '+' and '−'!

First, we will mention the geometric interpretation of classification. Consider a set of elements described by two continuous attributes as shown in figure 3. In the C4.5 method the training set is separated at a threshold of a continuous attribute. The effect of such a decision might be interpreted as an axis-parallel line in $\Re^2$ (see figure 3). Quite a lot of C4.5-type decisions are needed for correct separation of all training elements, although one line in general position would be enough as well. Our goal is to allow arbitrary figures like hyperplane, sphere and ellipsoid to separate elements of the training set.

In our new method the starting point is the pliant operator, which is a monotone weighted fuzzy operator from Dombi[6]:

$$c_\lambda(x,u;y,v) = \frac{1}{1+\left(u\left(\frac{1-x}{x}\right)^\lambda + v\left(\frac{1-y}{y}\right)^\lambda\right)^{\frac{1}{\lambda}}}$$

It is derived from the generalized form of mean values[7], which is closely related to the general form of fuzzy operators:

$$f^{-1}\left(\sum_{i=1}^{m} w_i f(x_i)\right), \text{ where } \sum_{i=1}^{m} w_i = 1.$$

The above mentioned operator is obtained from this general form when

$$f(x) = \left(\frac{1-x}{x}\right)^\lambda.$$

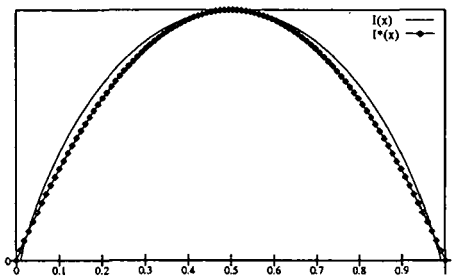


Figure 4: Entropy function for two classes and the new $4x(1-x)$ measure

We use the pliant operator when $u = v = \frac{1}{2}$ and $\lambda = 1$. In this special case it is easy to get the following measure of fuzziness:

$$I^*(x) = \frac{1}{c\left(\frac{1}{2}, \frac{1}{2}\right)} \cdot c(x, n(x)) = 4x(1-x).$$

This formula is much simpler than the entropy function and it behaves very similarly. The graph of the original and the new measure is shown on figure 4. It is easy to prove, that these two functions lead to the same result (in case of concept learning), since on interval $[0,1]$ both functions have maxima at $\frac{1}{2}$, minima at 0 and 1, they are strictly monotone on $[0, \frac{1}{2}]$ and on $[\frac{1}{2}, 1]$ and they never intersect each other. We are going to demonstrate their work on the following example, where the value of entropy function and the value of this new measure is calculated parallel.

This example, where the 14 elements of the training set is described by four attributes (two discrete and two continuous ones) and the examples belong to two classes, is taken from [1] (see table 1). First we have to calculate the entropy of the whole training set, that is

$$I(S) = -\frac{9}{14}\log\frac{9}{14} - \frac{5}{14}\log\frac{5}{14} = 0.9403.$$

With the new $I^*(x)$ measure we get

$$I^*(S) = 4 \cdot \frac{9}{14} \cdot \frac{5}{14} = 0.9184.$$

Then we try to separate the training set using the attribute 'outlook', so first we have to calculate the entropy of the three subsets (sunny, overcast, rain) $I(S_{sunny}) = -\frac{2}{5}\log\frac{2}{5} - \frac{3}{5}\log\frac{3}{5} = 0.9709$, $I(S_{overcast}) = -\frac{4}{4}\log\frac{4}{4} - 0 = 0$, $I(S_{rain}) = -\frac{3}{5}\log\frac{3}{5} - \frac{2}{5}\log\frac{2}{5} = 0.9709$ then the weighted sum of these values $E(outlook, S) = \frac{5}{14}0.9709 + \frac{4}{14}0 + \frac{5}{14}0.9709 = 0.6935$. Therefore $Gain(outlook) = 0.9403 - 0.6935 = 0.2468$. It says that we gain 0.2468 if we select attribute 'outlook' as a test.

If we apply the new measure we get the following: $I^*(S_{sunny}) = 4 \cdot \frac{2}{5} \cdot \frac{3}{5} = 0.96$, $I^*(S_{overcast}) = 4 \cdot \frac{4}{4} \cdot \frac{0}{4} = 0$, $I^*(S_{rain}) = 4 \cdot \frac{3}{5} \cdot \frac{2}{5} = 0.96$ and the weighted

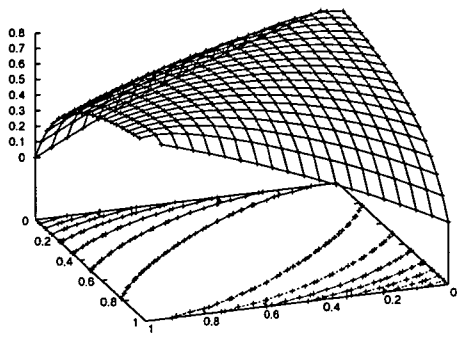


Figure 5: Test selection criteria

sum of these values is $E^*(outlook, S) = \frac{5}{14}0.96 + \frac{4}{14}0 + \frac{5}{14}0.96 = 0.6857$. Therefore $Gain^*(outlook) = 0.9184 - 0.6857 = 0.2327$. It is close to the previous result.

Now, we have to calculate the gain of discrete attribute 'windy' and the gain of continuous attributes 'temp' and 'humidity'. The selection criteria is maximization of gain, so finally we choose attribute 'outlook'. It is fairly a good choice, because nearly third of the training set is classified correctly. If we continue the algorithm on the first and third branch of the tree, we get the decision tree shown in figure 1.

5 The CDT method

In the previous section we introduced a new $I^*(x)$ measure instead of information entropy on the special case, if elements of training set belong to only two classes. Using this measure we are able to build up formulas similar to the ones in the ID3 method to compare the gain of possible tests. The result of our calculations for $E^*(t, S)$ is shown in figure 5. This works only for tests with two outcomes but decisions with geometric figures behave in this manner.

Our next problem is to describe the required geometric figures such as hyperplane, sphere and ellipsoid with bounded parameters. The geometric interpretation of classification shows that, if the training examples are described by n continuous attributes, the elements might be represented by points in the $\Re^n$ space. So we can calculate the bounding box of training examples as shown in figure 6. Denote R_{max} the distance of the farthest points of this bounding box from the center of it. Each point P in the sphere (with origin O and radius R_{max}) determines a hyperplane, with normal vector $\overrightarrow{OP}$ and one point P, that separates points of the training set:

$$f_s(\mathbf{p}, \mathbf{o}, \mathbf{x}) = \mathbf{n}(\mathbf{x} - \mathbf{p}) = \frac{\mathbf{p} - \mathbf{o}}{\|\mathbf{p} - \mathbf{o}\|_E}(\mathbf{x} - \mathbf{p}) = 0$$

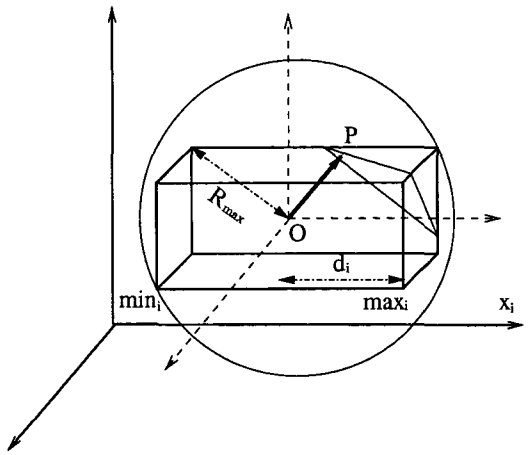


Figure 6: Description of hyperplane

where $p_i \in (O_i - R_{max}, O_i + R_{max})$. Other geometric figures may be described similarly.

Our last task is to count examples on one side of a geometric figure. Instead of taking strict bounds we use the well-known sigmoid function:

$$\sigma_1(\mathbf{x}, \mathbf{a}) = \frac{1}{1 + e^{-\lambda f(\mathbf{x}, \mathbf{a})}}$$

where $f(\mathbf{x}, \mathbf{a}) = 0$ describes the geometric figure (hyperplane, sphere or ellipsoid), vector $\mathbf{a}$ contains the parameters of the figure and λ gives the sharpness of the bound. This $\sigma_1(\mathbf{x}, \mathbf{a})$ function helps us counting examples belong to class C in one side of function $f(\mathbf{x}, \mathbf{a})$:

$$S_1^C(\mathbf{a}) = \frac{1}{|C|_T} \sum_{c(\mathbf{x}_i)=C} \sigma_1(\mathbf{x}_i, \mathbf{a}).$$

Now, we are ready to separate elements of the continuous space with arbitrary geometric figures, counting elements of the training set continuously (using the sigmoid function) and compare figures with different parameters (and compare different figures) using the *Gain*-type decision criteria. Finally, we should note that the parameters of the searched figures are found by global maximization of the *Gain* function, using stochastic global optimization technique with a specific local optimization method.

6 Examples

In this section we consider some two-dimensional examples to demonstrate the work of our new method introduced in the previous section. First, take the training set

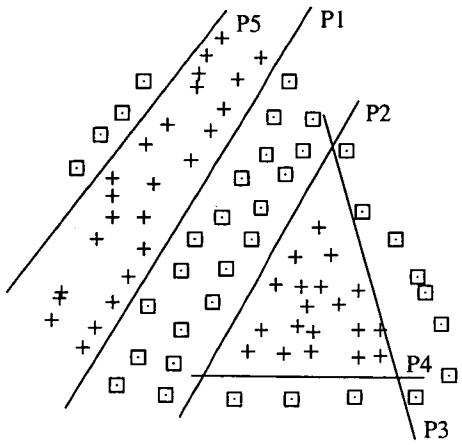


Figure 7: The extended triangle problem

shown in figure 7. This is the triangle problem extended with a few extra points. In this example we use only lines to separate elements of training set. The most interesting part of this example is the separation of the triangle. It is solved by three lines correctly, while in the C4.5 method (where only axis-parallel lines are allowed) about ten decisions are needed. (This example is also examined in [5]).

Another well-known hard task is the problem of two spirals on figure 8. The decision tree on the right side of the figure shows only the first few steps of the tree building method because this problem gives a large decision tree. As it can be seen, the elements of the training set have been quite well separated and the structure of the problem has already been realized in these very first steps. It is easy to imagine that the C4.5 method gives a very complex decision tree to solve this problem.

A further example could be a training set where most of the points are class $'+'$ and only a few points are class $'-'$ in a small group. Then the $'-'$ elements are taken out by a circle decision in our CDT method, while in C4.5 minimum four decisions are required to solve this problem.

7 Conclusions

Our new method is the generalization of the C4.5 algorithm in the following sense:

In C4.5 the continuous decision space is separated only with hypercubes whose edges are parallel with the coordinate axes, while in our new method arbitrary geometric figures like hyperplane, sphere and ellipsoid are allowed. The advantage of this improvement has been demonstrated on the previous examples. Of course

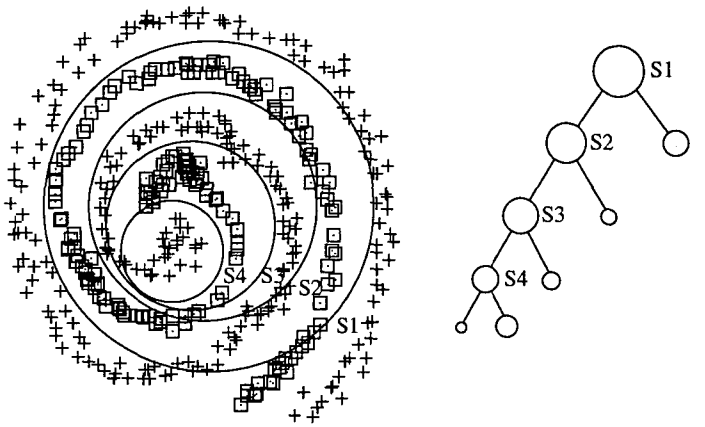


Figure 8: The problem of two spirals

it increases the computational complexity of the algorithm, but it also enlarges the variety of possible tests.

The entropy function is replaced with a simpler fuzzy measure (coming from a monotone fuzzy operator) and the test selection criteria is derived from this simpler measure. This simplification reduces the running time of the algorithm.

In our new method the elements of the training set are counted continuously, which means that the elements close to the bound of the figure belong to both regions (with different weights). Therefore, the decision criteria is a continuous function of the parameters of figure. This property makes the global optimization problem (finding parameters of geometric figure) simpler.

There are some restrictions in the current version of our new method: first, we use only continuous attributes. It is also possible to handle ordered discrete attributes but obviously it is impossible to handle unordered ones in this way. Second, we assumed, that all elements of the training set belong to two classes, so only concept learning is examined. Third, all the tests we use separate the training set into two smaller groups. Multivalued tests are impossible.

Finally, a few more improvements are going to be mentioned. First, it is worth to replace the general purpose global optimizer used in this method with a problem specific one, because it will enlarge the efficiency of the algorithm. Second, the running time on large datasets might be rather long, so in this case a sample of the training examples should be used to build up a tree. Our experiments shows that the algorithm is quite quick on a few hundred or thousand examples, but rather slow on larger (more hundred thousand elements) dataset. Third, sometimes the built up decision trees are too complex, so it is worth to apply some pre- or post-pruning[8, 9] method to simplify the result tree.

References

[1] J. R. Quinlan, *C4.5 Programs for Machine Learning*, Morgan Kaufmann Publishers, 1993.

[2] J. R. Quinlan, Improved Use of Continuous Attributes in C4.5, *Journal of Artificial Intelligence Research*, **4** (1996) 77-90.

[3] L. Hyafil and R. L. Rivest, Constructing optimal binary decision trees is NP-complete, *Information Processing Letters*, **5** (1976) 15-17.

[4] Stuart J. Russel and Peter Norvig, *Artificial Intelligence, a Modern Approach*, Prentice Hall, 1995.

[5] Krzysztof J. Cios and Ning Liu, A Machine Learning Method for Generation of a Neural Network Architecture: A Continuous ID3 Algorithm, *IEEE Transactions on Neural Networks*, **3** (1992) 280-290.

[6] Dombi J., A general class of fuzzy operators, the DeMorgan class of fuzzy operators and fuzziness measures induced by fuzzy operators, *Fuzzy Sets & Systems*, **8** (1982) 149-163.

[7] G. H. Hardy, J. E. Littlewood and G. Pólya, *Inequalities*, Cambridge University Press, 1934.

[8] Floriana Esposito, Donato Malerba and Giovanni Semeraro, A Comparative Analysis of Methods for Pruning Decision Trees, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **19** (1997) 476-491.

[9] Jim Kay, Comments on Esposito et al.,*IEEE Transactions on Pattern Analysis and Machine Intelligence*, **19** (1997) 492-493.

[10] Michael J. Kearns and Umesh V. Vazirani, *An Introduction to Computational Learning Theory*, The MIT Press, 1994.

[11] Tom M. Mitchell, *Machine Learning*, The McGrawn-Hill Companies, Inc., 1997.

Acta Cybernetica **15** (2001) 225–240.

Motion Planning Algorithms for Stratified Kinematic Systems with Application to the Hexapod Robot*

István Harmati† and Bálint Kiss†

Abstract

The paper addresses the motion planning problem of legged robots. Kinematic models of these robots are stratified, i.e. the equations of motion differ on different strata. An improved version of the motion planning algorithm proposed in the literature is compared with two alternative solutions via the example of the six-legged (hexapod) robot. The first alternative solution uses explicit integration of the vector fields while the second one exploits the flatness of a restricted subsystem.

1 Introduction

This paper addresses the motion planning problem (MPP) of kinematic systems with stratified configuration spaces [5].

Kinematic systems arise in the presence of non-holonomic constraints [8, 11]. A stratum is a submanifold of the configuration space. Strata may intersect each other leading possibly to a new stratum. We call the bottom stratum the submanifold with the lowest dimension (highest number of constraints). For stratified systems, the equations of motion differ for each submanifold and change discontinuously.

The problem arising in the control of such systems is that the bottom stratum is not controllable (in the sense that the Lie Algebra Rank Condition (LARC) [12, p. 367] is not satisfied). Therefore, one has to switch to other (higher dimensional) strata to find feasible trajectory between two points of the bottom stratum. Successive or cyclic switching between strata is called gaiting.

Since conventional motion planning algorithms (MPA) suppose smooth configuration spaces, they must be modified to work on stratified systems.

*Research partially supported by the Hungarian National Research Program under grant OTKA T 029072 and FKFP 0417/1997

†Department of Control Engineering and Information Technology, Budapest University of Technology and Economics, Pázmány Péter sétany 1/D, Budapest, H-1117 Hungary, Tel: (+361) 4632699, Fax: (+361) 4632204, e-mail: {harmati,bkiss}@seeger.iit.bme.hu

A general MPA, proposed by Lafferriere et al. [9] is adapted in [5, 6, 7] to solve the MPP for stratified systems. This algorithm uses piecewise constant inputs but is imprecise if the Lie algebra generated by the vector fields of the system fails to be nilpotent.

Motion planning can be easily solved for a restricted class of smooth kinematic systems which are differentially flat [3, 4, 10]. For such systems, the MPP is reduced to a simple interpolation problem in the space of the flat output since there is a one-to-one correspondence between sufficiently smooth trajectories of the flat output and feasible trajectories of the system.

Legged robotic structures are typical examples of stratified systems where the possible leg contact combinations define strata in the configuration space. The MPAs presented in the paper will be illustrated using the six-legged robot example where gaiting appears naturally during walking. The first algorithm is an improved version of the method of Lafferriere and Goodwine, the second one is a similar method but allows exact reaching of the final point and easy geometric management of the shape of the trajectory. The third algorithm uses the notion of flatness. The first algorithm is generally applicable while the second and third ones exploit the specific properties of the model and give better results.

The remaining part of the paper is organized as follows. The next section reviews some definitions used in the sequel. Section 3 presents the six-legged robot example in details. Generic methods known from the literature are briefly recalled and illustrated on the hexapod robot example in Section 4. We propose two alternative solutions for the MPP problem related to our example system in Section 5. Some concluding remarks close the paper. Simulation results illustrate all methods.

2 Definitions

The terminology of the differential geometry is used, the reader may refer to [1, 2, 13]. See [3, 4, 10] concerning the notion of differential flatness. We denote by ϕ_g^t the flow [1, p. 238] along the vector field g and by TM the tangent bundle [1, p. 159] of the manifold M.

Definition 1 (kinematic or driftless system). *A kinematic (or driftless) system is a dynamical system of the form:*

$$\dot{x} = \sum_{i=1}^{m} g_i(x) u_i \tag{1}$$

where $x \in \mathbb{R}^n$ and g_i $(i = 1, \ldots, m)$ are real analytic vector fields on $\mathbb{R}^n$.

Let $\mathcal{L}$ denote the Lie algebra generated by the vector fields $g_1, \ldots, g_m$. The system given by (1) is nilpotent with order k if $\mathcal{L}$ is nilpotent with the same order, i.e. $[v_1, [v_2, \ldots, [v_{k-1}, [v_k, v_{k+1}]] \ldots]]$ vanish for $v_i \in \{g_1, \ldots, g_m\}$, $i = 1, \ldots, k+1$.

Definition 2 (point to point MPP). *Consider the system given by (1). The point to point steering MPP consists of finding an input trajectory*

$t \to (u_1(t), \ldots, u_m(t))$ *that steers* (1) *from a given initial state* x_I *to a desired final state* x_F.

A MPA is a systematic procedure that provides an input trajectory (if it exists) solving the MPP for any given pair $(x_I, x_F) \in \mathbb{R}^n \times \mathbb{R}^n$.

Definition 3 (regularly stratified set). *A set* $\aleph \subset \mathbb{R}^n$ *defined by the union of smooth submanifolds of* $\mathbb{R}^n$ *(i.e. strata) is said to be a regularly stratified set.*

A stratum S_i is called lower (resp. higher) w.r.t. the stratum S_j if its dimension is lower (resp. higher). The stratum with the lowest dimension is the bottom stratum.

Definition 4 (moving on and moving off vector fields). *Let* $\aleph$ *be a regularly stratified set. Let* S_j *and* S_i *be two strata of* $\aleph$ *such that* S_i *is a submanifold of* S_j. *Let* $g \in TS_j$, *a vector field on* S_j. *If* $g|S_i \in TS_i$ *then* g *is a moving on vector field of* S_i, *otherwise* g *is a moving off vector field of* S_i.

Switching between strata is possible due to the existence of moving off vector fields. A moving on vector field is in general a moving off vector field for lower dimensional substrata.

Definition 5 (stratified kinematic system). *Let* $\aleph = \bigcup_{i=1}^{s} S_i$, $\aleph \subset \mathbb{R}^n$ *be a regularly stratified set where* S_i *are smooth submanifolds of* $\mathbb{R}^n$. *A stratified kinematic system with* $\aleph$ *as configuration space is given by a set of equations of motion, different on each stratum:*

$$x \in S_i \quad \Rightarrow \quad \dot{x} = \sum_{j \in I_i} g_{i,j}(x) u_j \qquad I_i \subset \{1, \ldots, m\} \qquad i = 1, \ldots s, \tag{2}$$

where I_i *is the set of the active inputs on the stratum* S_i *and such that the following are satisfied*

i. $g_{i,j}$ *are moving on vector fields of* S_i, *i.e.* $g_{i,j} \subset TS_i$

ii. $g_{l,j}, g_{k,j} \in TS_l \bigcap TS_k$ *for all* $x \in S_l \bigcap S_k$ *and all* $j \in I_k \bigcap I_l$

iii. If the system is on the strata S_i *(i.e.* $x \in S_i$*) at time* t *and* $j \notin I_i$ *then* $u_j(t) = 0$.
This means that the inputs remain always compatible with the stratification of the configuration space.

More than one set of equations of motion are defined for any point x *of the configuration space belonging to several strata. In this case the equation of motion is determined by the highest dimensional stratum whose tangent space contains the corresponding* $\dot{x}$ *and has moving off vector field w.r.t. to all lower dimensional substrata.*

Definition 6 (flatness). *The system* (1) *is differentially flat if one can find a set of variables (flat output)*

$$y = h(x, u, \dot{u}, \ddot{u}, \ldots, u^{(r)}), \qquad y \in \mathbb{R}^m$$

with r finite integer, such that

$$x = \alpha(y, \dot{y}, \ddot{y}, \ldots, y^{(q)}) \qquad u_i = \beta_i(y, \dot{y}, \ddot{y}, \ldots, y^{(q+1)}) \qquad i = 1, \ldots, m$$

with q a finite integer, and such that the system equations

$$\frac{d\alpha}{dt}(y, \dot{y}, \ddot{y}, \ldots, y^{(q+1)}) = \sum_{j=1}^{m} g_j(\alpha(y, \dot{y}, \ddot{y}, \ldots, y^{(q)})) \cdot \beta_j(y, \dot{y}, \ddot{y}, \ldots, y^{(q+1)})$$

are identically satisfied.

3 The hexapod robot example

Legged robots are stratified systems since their kinematics change discontinuously when a leg makes or brakes contact with the ground. The discussed motion planning methods are illustrated on the example of the hexapod robot (introduced in [5]), depicted in Figure 1 (left). The position and orientation of the body is given by

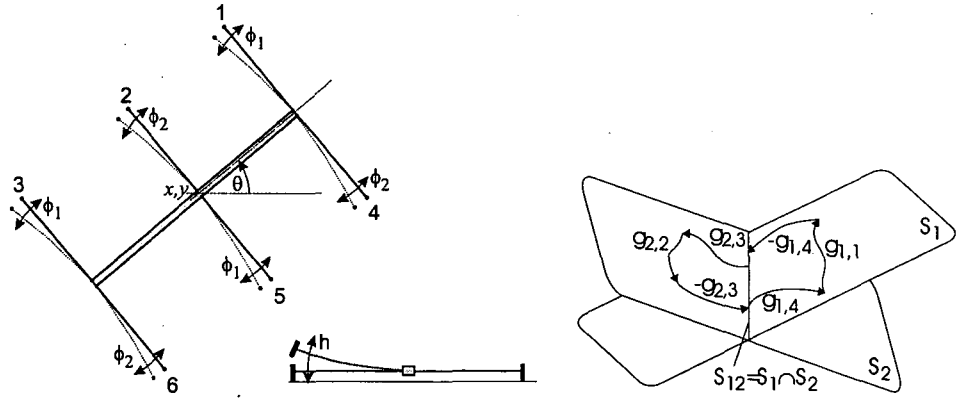


Figure 1: The hexapod robot (left); A tripod gait in the stratified configuration space of the hexapod robot (right)

the variables x, y and θ. The legs $\{1, 3, 5\}$ and the legs $\{2, 4, 6\}$ have the same leg angles (ϕ_1 and ϕ_2) and move (touch the ground and brake contact) simultaneously. The hexapod robot is assumed to be in stable position if the leg angles do not leave an admissible range given by $[\phi_{min}, \phi_{max}]$. The distance of the two set of legs from the ground is given by h_1 and h_2, respectively. Thus the configuration space is a regularly stratified set $\aleph$ in $M = \mathbb{R}^2 \times \mathbb{S}^3 \times \mathbb{R}^2$, where $\mathbb{S}$ is the unit circle. The state

vector is given by $\xi = (x, y, \theta, \phi_1, \phi_2, h_1, h_2)^T$. We have the following strata:

$$\begin{aligned}
S_1 &= \{\xi \in M : h_1 = 0\}, I_1 = \{1,2,4\}, u_3 = 0, \quad \text{legs } \{1,3,5\} \text{ on the ground} \\
S_2 &= \{\xi \in M : h_2 = 0\}, I_2 = \{1,2,3\}, u_4 = 0, \quad \text{legs } \{2,4,6\} \text{ on the ground} \\
S_{12} &= \{\xi \in M : h_1 = h_2 = 0\}, I_{12} = \{1,2\}, u_3 = u_4 = 0, \text{all legs on the ground}
\end{aligned}$$

The robot is able to perform different kind of motions:

1. If only one of the set of legs $\{1,3,5\}$ (resp. $\{2,4,6\}$) touches the ground (active legs) while the other set of legs remains in the air (inactive legs) then the body may move by changing the angles ϕ_1 (resp. ϕ_2). In this case, only the active legs have influence on the planar motion of the robot, but the inactive legs may also change their leg angles. The successive commutation of the active and inactive legs (called gaiting) makes the walking possible.
2. If all legs touch the ground at the same time then all legs are active. The angles ϕ_1 and ϕ_2 may change. This motion will be referred to as the paddling motion.

These motions are possible along the vector fields:

$$\begin{aligned}
g_{12,1}(\xi) &= \begin{pmatrix} \cos\theta & \sin\theta & l & 1 & 0 & 0 & 0 \end{pmatrix}^T \\
g_{12,2}(\xi) &= \begin{pmatrix} \cos\theta & \sin\theta & -l & 0 & 1 & 0 & 0 \end{pmatrix}^T \\
g_{1,1}(\xi) &= g_{12,1}(\xi) \\
g_{1,2}(\xi) &= \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}^T \\
g_{1,4}(\xi) &= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}^T \\
g_{2,1}(\xi) &= \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}^T \\
g_{2,2}(\xi) &= g_{12,2}(\xi) \\
g_{2,3}(\xi) &= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}^T
\end{aligned} \tag{3}$$

where l denotes the length of the legs ($l = 0.1$m is used for simulations). The equations of motion in the different strata read:

$$\xi \in S_1 \Rightarrow \quad \dot{\xi} = g_{1,1}(\xi)u_1 + g_{1,2}(\xi)u_2 + g_{1,4}(\xi)u_4 \tag{4a}$$

$$\xi \in S_2 \Rightarrow \quad \dot{\xi} = g_{2,1}(\xi)u_1 + g_{2,2}(\xi)u_2 + g_{2,3}(\xi)u_3 \tag{4b}$$

$$\xi \in S_{12} \Rightarrow \quad \dot{\xi} = g_{12,1}(\xi)u_1 + g_{12,2}(\xi)u_2 \tag{4c}$$

Thus the moving on vector fields of the (bottom) stratum S_{12} are $g_{12,1}$ and $g_{12,2}$. Since $S_{12} = S_1 \bigcap S_2$, the moving on vector fields $g_{1,4}$ and $g_{2,3}$ of S_1 and S_2, respectively, are moving off vector fields of S_{12} (lift legs $\{1,3,5\}$ or legs $\{2,4,6\}$).

Remark 1. *Since the leg heights h_1 and h_2 do not change along the integral curves of the vector fields $g_{1,2}$ and $g_{2,1}$ (the last two components of these vector fields vanish) we have that $g_{1,2}(\xi) \in TS_{12}$ and $g_{2,1}(\xi) \in TS_{12}$ for all $\xi \in S_{12}$, although $g_{1,2}$ and $g_{2,1}$ are defined on S_1 and S_2, respectively.*

Definition 7 (tripod gait). *A flow sequence of the form*

$$\xi_F = \underbrace{\phi_{-g_{2,3}}^{t_8}}_{S_{12}\leftarrow S_2} \circ \underbrace{\phi_{g_{2,2}}^{t_7}}_{on\ S_2} \circ \underbrace{\phi_{g_{2,1}}^{t_6}}_{on\ S_2} \circ \underbrace{\phi_{g_{2,3}}^{t_5}}_{S_2\leftarrow S_{12}} \circ \underbrace{\phi_{-g_{1,4}}^{t_4}}_{S_{12}\leftarrow S_1} \circ \underbrace{\phi_{g_{1,2}}^{t_3}}_{on\ S_1} \circ \underbrace{\phi_{g_{1,1}}^{t_2}}_{on\ S_1} \circ \underbrace{\phi_{g_{1,4}}^{t_1}}_{S_1\leftarrow S_{12}} (\xi_I), \qquad (5)$$

such that $\xi_F, \xi_I \in S_{12}$ is called a tripod gait.

The stratification of the configuration space of the hexapod robot with the flow during a tripod gait is illustrated in Figure 1 (right).

The MPP related to the hexapod robot consists of finding control inputs $t \to u_i(t)$, $i = 1, \ldots, 4$ such that the generated trajectory connects the points p and q in the *bottom stratum* S_{12} with $p = (x_I, y_I, \theta_I, \phi_{1,I}, \phi_{2,I})^T$ to $q = (x_F, y_F, \theta_F, \phi_{1,F}, \phi_{2,F})^T$.

4 Generic methods

4.1 An MPA for smooth kinematic systems

Based on the paper of Lafferriere and Sussmann [9] we present first a motion planning algorithm for the smooth kinematic system (1) which will be extended for stratified systems in the following subsection using [5]. We assume that (1) is completely controllable [12, p. 367]. Suppose that we want to find a trajectory connecting p to q. The algorithm proposed in [9] is the following.

Algorithm 1:

Step I. Extend the system (1) to

$$\dot{x} = v_1 g_1(x) + \ldots + v_m g_m(x) + v_{m+1} g_{m+1}(x) + \ldots + v_r g_r(x) \qquad (6)$$

where the vector fields $g_{m+1}, \ldots, g_r$ $(r \geq n)$ are defined by higher order Lie brackets of g_i, $i = 1, \ldots, m$, selected such that $\text{span}\{g_1(x), \ldots, g_r(x)\} = \mathbb{R}^n$ (This is always possible due to the controllability assumption.)

Step II. Find a control v that steers the extended system (6) from p to q. Since the vector fields $\{g_1(x), \ldots, g_r(x)\}$ span $\mathbb{R}^n$, one can choose a straight line segment between p and q. The corresponding input v of (6) is obtained by matrix inversion.

Step III. We compute a control u for the original system (1) that substitutes the control v of the extended system (6). The substitution means that the trajectory of (1) obtained by applying u connects the same initial and final points p and q, as the straight line trajectory of the extended system (6) under the action of v. This is done using the following steps:

Step 1. We calculate the order of nilpotency of the Lie algebra $\mathcal{L}$ associated to system (1). If the order of nilpotency is not finite, then we use a kth order finite approximation of the Lie algebra $\mathcal{L}$ by replacing the brackets $[v_1, [v_2, \ldots, [v_{k-1}, [v_k, v_{k+1}]] \ldots]]$ by zero for $v_i \in \{g_1, \ldots, g_m\}$, $i = 1, \ldots, k+1$.

Step 2. We determine the Philip Hall basis of $\mathcal{L}$ [9, § 6], [11, p. 704]. The elements of this basis are symbolic Lie brackets of the vector fields $g_1, \ldots, g_m$.

Step 3. We solve the formal differential equation [12]

$$\dot{S}(t) = S(t)\left(\sum_{j=1}^{r} v_j g_j\right) \qquad S(0) = 1, \tag{7}$$

where the solution $S(t)$ is an element of a special nilpotent Lie group [9, § 7] and has the form

$$S(t) = e^{\tilde{h}_1(t)B_1} \ldots e^{\tilde{h}_{s-1}(t)B_{s-1}} e^{\tilde{h}_s(t)B_s}, \tag{8}$$

where B_i are the elements of the P. Hall basis and $\tilde{h}_i(t)$ are the forward P. Hall coordinates. The symbolic expression of $S(t)$ gives a sequence of flows along the vector fields of the P. Hall basis of $\mathcal{L}$.

Step 4. Let $q = S(1) \cdot p$. Introduce $\bar{h}_i = \tilde{h}(1)$ for $(i = 1, \ldots, r)$. Then we get

$$q \approx e^{\bar{h}_1 B_1} \ldots e^{\bar{h}_{s-1} B_{s-1}} e^{\bar{h}_s B_s} \cdot p. \tag{9}$$

where we have equality if no nilpotent approximation is needed.

Step 5. Due to the nilpotency of $\mathcal{L}$ (or to its nilpotent approximation, see Step 1), the Campbell-Baker-Hausdorff formula [13, p. 114] allows to replace the flows along basis elements obtained by Lie brackets with a sequence of flows along the vector fields $g_1, \ldots, g_m$. This gives

$$q \approx \left(\prod_{i=1}^{l<\infty} e^{\chi_i X_i}\right) p \qquad X_i \in \{g_1, \ldots, g_m\} \tag{10}$$

where χ_i are constant.

Step 6. Using (10), the piecewise constant input of the original system in the time interval $i - 1 < t < i$, $i = 1, \ldots, l$) is given by

$$\begin{array}{lll} u_j(t) = 0 & \text{if} & X_i \neq g_j \\ u_j(t) = \chi_i & \text{if} & X_i = g_j \end{array} \qquad j = 1, \ldots, m. \tag{11}$$

Note that the trajectories along which the original system (1) and the extended system (6) arrive to q are, in general, different.

Remark 2. *The motion planning algorithm gives trajectories such that only one of the inputs is nonzero at the same time. The exact shape of the trajectory can not be planned explicitly.*

Remark 3. *In practice, if the order of nilpotency is too high for numeric calculations, then one may use again a lower order approximation forcing the high order brackets to vanish similarly to the case of non-nilpotent systems.*

4.2 Application to stratified systems

By Definition 5, different strata give different equations of the type (1). Since the controllability on a given stratum is not guaranteed, Algorithm 1 can not be directly applied. To overcome this difficulty, vector fields from higher strata are also considered.

Detailed presentation can be found in [5, 6, 7]. Let us directly turn our attention to the hexapod robot example. Note that the system is not controllable on S_{12} because the Lie algebra generated by the moving on vector fields ($g_{12,1}$ and $g_{12,2}$) of the bottom stratum does not satisfy the LARC.

Lemma 1. *For any sequence of flows along the vector fields $g_{12,1}$, $g_{12,2}$, $g_{2,1}$, and $g_{1,2}$ in the bottom stratum S_{12}, connecting p to q, there exists a sequence of tripod gaits connecting p to q.*

Proof. Without loss of generality, consider the following flow sequence in the bottom stratum:

$$\xi_F = \phi^{t_4}_{g_{12,2}} \circ \phi^{t_3}_{g_{2,1}} \circ \phi^{t_2}_{g_{1,2}} \circ \phi^{t_1}_{g_{12,1}}(\xi_I) \tag{12}$$

Using (3), replace $\phi^{t_4}_{g_{12,2}}$ by $\phi^{t_4}_{g_{2,2}}$ and $\phi^{t_1}_{g_{12,1}}$ by $\phi^{t_1}_{g_{1,1}}$:

$$\xi_F = \phi^{t_4}_{g_{2,2}} \circ \phi^{t_3}_{g_{2,1}} \circ \phi^{t_2}_{g_{1,2}} \circ \phi^{t_1}_{g_{1,1}}(\xi_I). \tag{13}$$

Insert the flows sequences $\phi^{t_5}_{-g_{2,3}} \circ \phi^{t_5}_{g_{2,3}} = I$ and $\phi^{t_6}_{-g_{1,4}} \circ \phi^{t_6}_{g_{1,4}} = I$:

$$\xi_F = \phi^{t_5}_{-g_{2,3}} \circ \phi^{t_5}_{g_{2,3}} \circ \phi^{t_4}_{g_{2,2}} \circ \phi^{t_3}_{g_{2,1}} \circ \phi^{t_2}_{g_{1,2}} \circ \phi^{t_1}_{g_{1,1}} \circ \phi^{t_6}_{-g_{1,4}} \circ \phi^{t_6}_{g_{1,4}}(\xi_I). \tag{14}$$

Since $[g_{1,4}, g_{1,1}] = 0$, $[g_{1,4}, g_{1,2}] = 0$, $[g_{2,3}, g_{2,1}] = 0$ and $[g_{2,3}, g_{2,2}] = 0$, the flows along these pairs of vector fields commute. This allows to rearrange the terms to obtain:

$$\xi_F = \phi^{t_5}_{-g_{2,3}} \circ \phi^{t_4}_{g_{2,2}} \circ \phi^{t_3}_{g_{2,1}} \circ \phi^{t_5}_{g_{2,3}} \circ \phi^{t_6}_{-g_{1,4}} \circ \phi^{t_2}_{g_{1,2}} \circ \phi^{t_1}_{g_{1,1}} \circ \phi^{t_6}_{g_{1,4}}(\xi_I), \tag{15}$$

which is a tripod gait by Definition 7. □

Based on the previous lemma and Algorithm 1, the following algorithm is proposed to solve the MPP of the hexapod robot being a stratified kinematic system [5, 6].

Algorithm 2:

Step I. Construct a smooth kinematic system on S_{12} from the vector fields $g_{12,1}$, $g_{12,2}$, $g_{1,2}$ and $g_{2,1}$ referred to as the bottom stratified system:

$$\dot{\xi} = g_{12,1}(\xi)w_1 + g_{12,2}(\xi)w_2 + g_{1,2}(\xi)w_3 + g_{2,1}(\xi)w_4 \quad \xi \in S_{12}, \tag{16}$$

where $w_1, \ldots, w_4$ is the input vector. Note that this system is controllable in S_{12}.

Step II. Use Algorithm 1 on the bottom stratified system (16), to obtain the piecewise constant input sequence which makes possible to connect p to q. Since the Lie algebra associated to (16) fails to be nilpotent, a nilpotent approximation has to be used. The solution, which reaches a point $\tilde{q}$ close to q, gives a sequence of flows along the vector fields $g_{12,1}$, $g_{12,2}$, $g_{1,2}$ and $g_{2,1}$:

$$q \approx \tilde{q} = \phi_{X_s}^{\chi_s} \circ \phi_{X_{s-1}}^{\chi_{s-1}} \circ \cdots \phi_{X_1}^{\chi_1}(p) \quad X_i \in \{g_{12,1}, g_{12,2}, g_{1,2}, g_{2,1}\}. \tag{17}$$

Step III. Using Lemma 1, one can find a gaiting sequence by inserting flows of moving off vector fields of the strata S_{12} where the successive flows are in different strata:

$$q \approx \tilde{q} = \phi_{X_l}^{\chi_l} \circ \cdots \phi_{X_1}^{\chi_1}(p) \quad X_i \in \{g_{12,1}, g_{12,2}, g_{1,1}, g_{1,2}, g_{1,4}, g_{2,1}, g_{2,2}, g_{2,3}\}, \tag{18}$$

with $l > s$ due to the insertions.

Step IV. Let T be the desired travelling time between p and q. Introduce $T_s = \sum_{i=1}^{l} \chi_i$. In order to arrive to $\tilde{q}$ at time T, the inputs and the switching time are obtained from (18) as

$$u_1(t) = \begin{cases} \frac{T_s}{T}\chi_i & \text{if } X_i \in \{g_{12,1}, g_{1,1}, g_{2,1}\} \\ 0 & \text{otherwise} \end{cases} \qquad u_3(t) = \begin{cases} \frac{T_s}{T}\chi_i & \text{if } X_i = g_{2,3} \\ 0 & \text{otherwise} \end{cases}$$

$$u_2(t) = \begin{cases} \frac{T_s}{T}\chi_i & \text{if } X_i \in \{g_{12,2}, g_{1,2}, g_{2,2}\} \\ 0 & \text{otherwise} \end{cases} \qquad u_4(t) = \begin{cases} \frac{T_s}{T}\chi_i & \text{if } X_i = g_{1,4} \\ 0 & \text{otherwise,} \end{cases} \tag{19}$$

for the time interval $(i-1)\frac{T}{T_s} < t < i\frac{T}{T_s}$, $i = 1, \ldots, l$.

If the bottom stratified system is not nilpotent, then the approximation error can be decreased by inserting intermediate reference points between p and q. The section between two neighboring intermediate reference points is called a subsegment. Then the above algorithm can be applied successively for the subsegments. The major problem of the method is the choice of the length of the subsegments along the trajectory. There exists a critical distance under which the algorithm converges (see [9]), however its estimation is a hard question without theoretical answer. The proposal of [6, 7] can be used modifying iteratively the appropriate length during the computation.

4.3 Software implementation and simulation

Let us illustrate the preceding stratified MPA (Algorithm 2) with a simulation software[1] written using the improvements of [6, 7], which can be also exploited for more complex problems (e.g. dextrous manipulation with robotic hands). The simulation results for the hexapod robot with two different prescribed orientations are

[1] All simulation programs are written using Matlab and the Symbolic Math Toolbox.

given in Figure 2 (left: constant orientation, right: tangent orientation). Figure 2 shows the reference points and the developed path of the hexapod robot, the orientation is not illustrated. The x, y coordinate axes are scaled in m. The trajectories are specified using the same sequence of reference points in the x, y plane but the orientations at the reference points are different. It can be seen that in spite of the

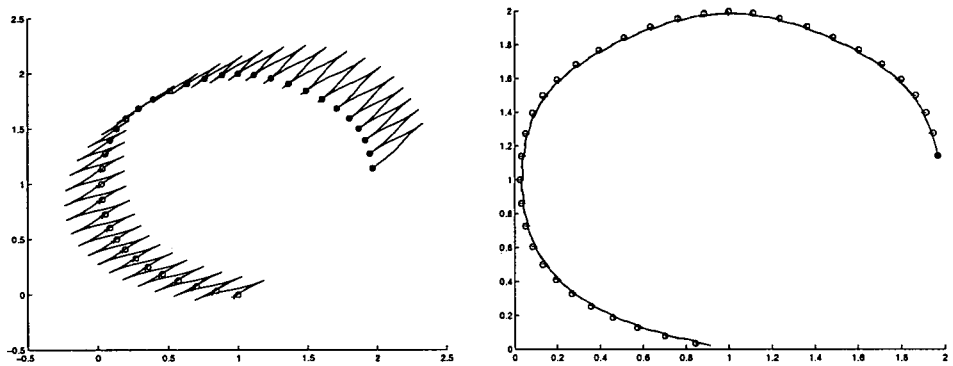


Figure 2: Stratified motion planning for hexapod robot: i) constant orientation, $\theta = \frac{\pi}{3}$radian (left); ii) tangent orientation; "o": reference points, "+": reached points (right)

fact that the Lie algebra $\mathcal{L}$ of the vector fields of (3) fails to be nilpotent, the solution based on a nilpotent approximation of order 1 gives satisfactory accuracy for both cases. If the desired orientation is close to the tangent of the curve connecting the reference points, then the algorithm has less difficulty to provide more precise solution.

5 Two alternative MPP solutions

5.1 Exact reaching along circle arcs with piecewise constant inputs

Assume again, as proposed in [9], that the inputs are piecewise constant and all but one input vanish at the same time.

First, observe that the flows along the vector fields $g_{12,1}$ (resp. $g_{12,2}$) are circles in the x, y plane. (Recall that in S_1 (resp. in S_2) the input u_3 (resp. u_4) vanish.) To see this, integrate $g_{12,1}$ with the initial conditions $\theta(0) = \theta_0$, $x(0) = x_0$, $y(0) = y_0$

and with $u_1(t) = \pm u_0$ (with $u_0 > 0$ and constant). We obtain

$$\begin{aligned}\theta(t) &= \pm l u_0 t + \theta_0 \\ y(t) &= -\frac{1}{l}\cos(\pm l u_0 t + \theta_0) + \frac{1}{l}\cos\theta_0 + y_0 \\ x(t) &= \frac{1}{l}\sin(\pm l u_0 t + \theta_0) - \frac{1}{l}\sin\theta_0 + x_0,\end{aligned}$$

which leads to

$$\left(y(t) - \frac{1}{l}\cos\theta_0 - y_0\right)^2 + \left(x(t) + \frac{1}{l}\sin\theta_0 - x_0\right)^2 = \frac{1}{l^2},$$

giving a circle. Integrating $g_{12,2}$ with constant input $u_2(t) = \pm u_0$ gives similar result. Consequently, if we restrict ourselves to trajectories along the flows $g_{12,1}$ and $g_{12,2}$ with constant velocities, the robot may follow arcs of radius $\frac{1}{l}$. Given a position and an orientation (x, y, θ) of the hexapod robot in the plane, an integral curve of the vector field $g_{12,1}$ passing through (x, y, θ) is referred to as an arc of type 1 and an integral curve of the vector field $g_{12,2}$ as an arc of type 2.

Definition 8 (Admissible trajectory). *A sequence of arcs in the plane is said to be an admissible trajectory for the hexapod robot if each two subsequent arcs, a_i and a_{i+1} have a unique common point where the tangent directions to both arcs coincide and, additionally, a_i and a_{i+1} are of different type.*

A simple trajectory planning method that constructs an admissible trajectory between two points $p = (x_I, y_I, \theta_I, \phi_{1,I}, \phi_{2,I})^T$ and $q = (x_F, y_F, \theta_F, \phi_{1,F}, \phi_{2,F})^T$ is based on the construction of tangent circles.

Algorithm 3:

Step I. Determine the flows (circles in the x, y plane) of type 1 and 2 passing through the point x_I, y_I and choose the one with the origin closer to x_F, y_F. If the distances of the origins of the two circles from the point x_F, y_F are the same, chose the circle of type one. Denote this circle by C_I.

Step II. Determine the flows (circles in the x, y plane) of type 1 and 2 passing through the point x_F, y_F and choose the one with the origin closer to the that of C_I. If the distances of the origins of the two circles from that of C_I are the same, chose the circle of type one. Denote this circle by C_F.

Step III. Calculate the unit vector

$$v = \frac{1}{\sqrt{(x_F - x_I)^2 + (y_F - y_I)^2}} \begin{bmatrix} x_F - x_I \\ y_F - y_F \end{bmatrix}$$

and let $\mathcal{C} = \{C_I\}$ a list of circles with unique element C_I.

Step IV. Get the last circle from list $\mathcal{C}$ and denote its origin by (x_C, y_C) and its type by $h \in \{1, 2\}$. Append to the list $\mathcal{C}$ a circle of radius $\frac{1}{l}$ with origin

$$\begin{bmatrix} x_C \\ y_C \end{bmatrix} + \frac{2}{l} v$$

and let the type of this circle be different of h. Repeat this step until *both* of the following conditions are fulfilled:

- the distance of the origin of the newly appended circle from the origin of C_F is less or equal to $\frac{4}{l}$,
- the type of the newly appended circle is the same than that of C_F.

Step V. Construct a circle which is tangent to the last circle of the list $\mathcal{C}$ and to C_F and let its type be different from that of C_F. Append this circle to the list $\mathcal{C}$. Append C_F to the list $\mathcal{C}$.

Step VI. The reference trajectory is given by the sequence of arcs of the tangent circles of the list $\mathcal{C}$.

Step VII. The contact points between the tangent circles allow to calculate the change of orientation (the evolution of θ) along each arc.

Step VIII. Since the input is constant along each arc the travelling time belonging to every arc can be obtained from the equation $\dot{\theta}(t) = lu_1(t) - lu_2(t)$. The travelling time T_i along the ith circle arc can be expressed from

$$\frac{\theta_i - \theta_{i-1}}{T_i} = lu_1(t) - lu_2(t). \tag{20}$$

Step IX. The travelling time along the arcs allow to calculate the variation of the leg angles ϕ_i, $i = 1, 2$, and thus the moments where ϕ_i leaves its admissible range $[\phi_{min}, \phi_{max}]$. These moments specify the gaits when the corresponding legs are needed to be lift off and rotated back by insertion of moving off vector fields ($g_{1,4}$ and $g_{2,3}$ to lift off and put down the legs).

Note that the recurrence defined in **Step IV** always terminates after a finite number of iterations if the distance of the initial and the final point is finite. Note also that the construction of the circle in **Step V** is always possible since the distance of the origins of the last circle of $\mathcal{C}$ and C_F is less than $\frac{4}{l}$.

The algorithm guarantees the exact reach of the final point. The shape of the trajectory is in direct control (because it consists of simple curve pieces, i.e. arcs) and this makes possible to extend the algorithm with obstacle avoidance.

The path constructed by the algorithm is optimal neither in length nor in number of gaits. In fact, by appending shorter sections of arcs, the straight line trajectory, optimal in length, can be more closely approximated. Moreover, since the legs have to be rotated back periodically anyway, gaits at those moments are not prohibitive and could allow shorter arcs than half-circles inserted by our algorithm. The deeper analysis of the questions of finding paths which are optimal in terms of gaits or length is beyond the scope of the present paper.

Figure 3 gives the trajectory connecting $p = (-50, -10, 0, 0, 0)$ and $q = (5, -5, -\frac{3\pi}{4}, 0, 0)$. The velocity along the flows $g_{12,1}$ and $g_{12,2}$ is $u_i = \pm u_0 = \pm 5$, $i = 1, 2$.

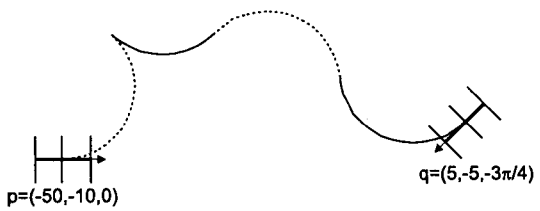


Figure 3: Trajectory along arcs of circles

5.2 Exact reaching along sufficiently smooth curve in the x, y plane using paddling motion

Proposition 1. *The kinematics of the bottom stratum* (4c)*, restricted to the tangent space of* (x, y, θ)*, and given by the equations*

$$\dot{x} = \cos\theta(u_1 + u_2) \tag{21a}$$

$$\dot{y} = \sin\theta(u_1 + u_2) \tag{21b}$$

$$\dot{\theta} = l(u_1 - u_2), \tag{21c}$$

is differentially flat. A flat output is the position of the robot: (x, y).

Proof. From the first two equations one calculates the trajectory of θ using $\dot{x}$, $\dot{y}$:

$$\theta = \arctan 2(\dot{y}, \dot{x}), \tag{22}$$

where arctan 2 is the inverse function of tan which gets its value in the interval $(-\pi, \pi]$. Then, using (21a) or (21b) together with (21c), one calculates u_1 and u_2 as function of $\dot{x}, \dot{y}, \theta, \dot{\theta}$. But θ is in turn function of $\dot{x}, \dot{y}$. Consequently, all variables are functions of $\dot{x}, \ddot{x}, \dot{y}, \ddot{y}$, the successive time derivatives of x and y which proves that the system is flat with the flat output (x, y) as claimed. □

Flatness implies that the (sufficiently smooth) trajectory of the position determines completely the trajectory of the remaining variables of the restricted model (θ, u_1, and u_2). In the case of the hexapod robot, the desired trajectory in the x, y plane must be continuously differentiable.

The variables ϕ_1 and ϕ_2 are not included in the restricted model (21). However, their trajectory and the gait moments can be obtained by integration of u_1 and u_2 using the relations $\dot{\phi}_1 = u_1$, $\dot{\phi}_2 = u_2$. Their desired final values $(\phi_{1,F}, \phi_{2,F})$ can be reached by lifting off and rotating the corresponding legs which leaves unchanged the position and orientation of the hexapod robot. This gives the following MPA:

Algorithm 4:

Step I. Let the reach time be fixed to T. Construct at least third order polynomials for $x(t)$ and y(t) whose coefficients are determined using the constraints

$$x(0) = x_I \qquad x(T) = x_F \qquad y(0) = y_I \qquad y(T) = y_F$$

$$\dot{x}(0) = \cos\theta_I \qquad \dot{x}(T) = \cos\theta_F \qquad \dot{y}(0) = \sin\theta_I \qquad \dot{y}(T) = \sin\theta_F.$$

Step II. Calculate the trajectory of θ, u_1, and u_2 as a function of $x(t), y(t)$ and their time derivatives. This is possible since the corresponding subsystem is flat.

Step III. Integrate numerically u_i $(i = 1, 2)$ to obtain ϕ_i. Each time when ϕ_i goes out of range insert a gait which lifts off and rotates back the corresponding legs. When the desired position and orientation is reached, lift off the legs and rotate them until the desired final leg angles are reached.

Remark 4. *Choosing T too small results large u_i. If there are limitations on u_i, they can be respected by re-scaling the time along the trajectory (control of the clock), and/or by increasing T.*

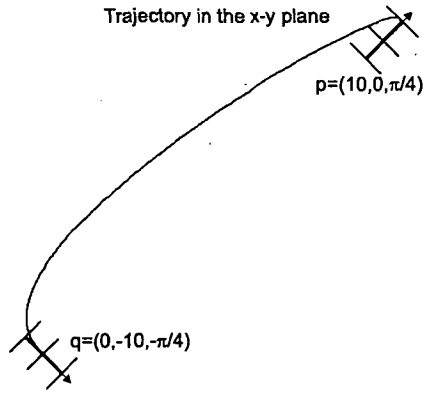


Figure 4: Motion planning along arbitrary curve (paddling motion) using flatness.

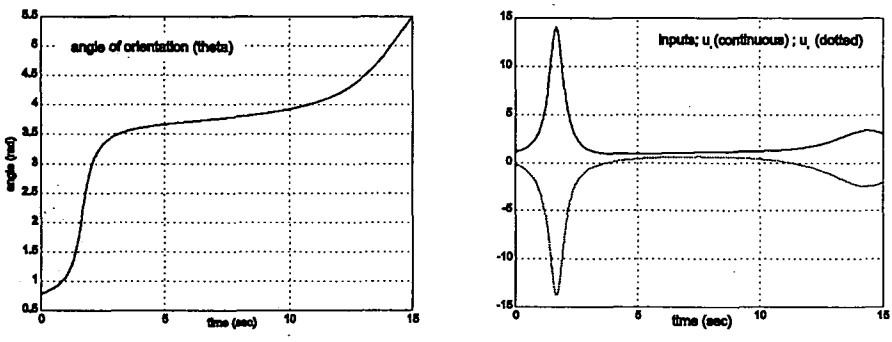


Figure 5: Trajectory of the orientation and the inputs using paddling motion.

A trajectory connecting $p = (10, 0, \pi/4)$ and $q = (0, -10, -\pi/4)$ is illustrated in Figure 4. The evolution of the orientation angle θ and the input trajectory are given in Figure 5.

6 Conclusion

The kinematics of walking robots give rise to stratified systems. We presented three algorithms solving the steering MPP of the hexapod robot. It turns out that the generic methods can be improved if the particular geometry of the given system is exploited which is the case of the last two MPAs. However, these alternative methods (Algorithms 3 and 4) use the specific properties of the robot model, thus they cannot be widely generalized.

The first alternative method presented in Subsection 5.1 using simple geometric manipulation is well adapted to real-time trajectory generation since it involves neither numerical integration nor optimization.

Both proposed alternative methods are able to reach exactly the desired final point. This is not possible (without the use of feedback techniques) for non-nilpotent systems (e.g. the hexapod robot, considered in the paper) using the generic methods. The additional advantages are that the alternative algorithms work precisely without the insertion of additional reference points between the starting and final points.

The motion planning problem discussed in this paper works on a kinematic model of the hexapod robot. The extension to a model incorporating friction effects at the leg contacts, more complicated leg kinematics, the dynamics of the legs and that of the robot itself needs to extend existing MPAs from driftless systems to a more general class of nonlinear control mechanical systems. The problem remains tractable if the extended model remains flat, but is a difficult one for general stratified mechanical systems and is still an open research area.

References

[1] R. Abraham, J. E., Marsden, and T. Ratiu. *Manifolds, tensor analysis and applications.* Springer - Verlag New York Inc., 1988.

[2] D. V. Alekseevskij, A. M. Vinogradov, and Lychagin V. V. *Geometry I - Basic Ideas and Concepts of Differential Geometry*, volume 28 of *Encyclopedia of Mathematical Sciences.* Springer - Verlag New York Inc., 1991.

[3] M. Fliess, J. Lévine, Ph. Martin, and P. Rouchon. Flatness and defect of nonlinear systems: introductory theory and examples. *International Journal of Control*, 61(6):1327–1361, 1995.

[4] M. Fliess, J. Lévine, Ph. Martin, and P. Rouchon. A Lie Bäcklund approach to equivalence and flatness of nonlinear systems. *IEEE Transactions on Automatic Control*, 38:700–716, 1999.

[5] B. Goodwine. *Control of stratified systems with robotic applications.* PhD thesis, California Institute of Technology, 1998.

[6] I. Harmati, B. Lantos, and S. Payandeh. Extensions of nonlinear control concept to object manipulation with finger relocation. In *Proceedings of the 6th International IFAC Symposium on Robot Control*, pages 223–228, Vienna, Austria, 21-23 September, 2000.

[7] I. Harmati, B. Lantos, and S. Payandeh. Improved stratified control for hexapod robots and object manipulation with finger relocation. *In Press. Periodica Polytechnica, Budapest, Hungary*, 2000.

[8] S. D. Kelly and R. M. Murray. Geometric phases and robotic locomotion. *Journal of Robotic Systems*, 12(6):417–431, 1995.

[9] G. Lafferriere and H. J. Sussmann. Motion planning for controllable systems without drift. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1148–1153, Sacramento, CA, USA, 1991.

[10] Ph. Martin and P. Rouchon. Feedback linearization and driftless systems. *Mathematics of Control Signals, and Systems*, 7(3):235–254, 1994.

[11] R. M. Murray and S. S. Sastry Nonholonomic Motion Plannaing: Steering Using Sinusoids. *IEEE Transactions on Automatic Control*, 38(5):700–716, 1993.

[12] H. J. Sussmann, New differential geometric methods in nonholonomic path finding *Systems, Models, and Feedback: Theory and Applications*, A. Isidori and T. J. Tarn Eds, Birkhäuser, Boston, pp. 365-384., 1992

[13] V. Varadarajan, *Lie Groups, Lie Algebras, and Their Representations.* Springer-Verlag, 1994.

Acta Cybernetica **15**(2001) 241–256.

SmallSteps: An Adaptive Distance-based Clustering Algorithm

Gy. Koch* and J. Dombi*

Abstract

In this article we propose a new distance-based clustering algorithm. Distance-based clustering methods operate on data sets that are in similarity space, where the similarities/dissimilarities between the objects are given by a matrix. These algorithms have at least $O(n^2)$ time complexity, where n is the number of objects. One of the latest distance-based method is Chameleon which, according to experiences, works well only on larger data sets and fails on relatively smaller ones. This contraditcs the fact that the $O(n^2)$ time complexity makes the distance-based algorithms unsuitable for huge data sets. Thus we developed a new distance-based method (SmallSteps), which can handle relatively small amount of objects too. In our solution we are looking for connected graphs which have edges with a maximum weight computed on the environments of the objects. The method is capable to detect clusters with different shapes, sizes or densities, it is able to automatically determine the number of clusters and has a special ability to divide clusters into subclusters.

1 Introduction

Clustering is one of the most commonly used statistical methods. It can be seen as an unsupervised machine learning process where the algorithm has to divide a set of objects ($X = \{x_1, \ldots, x_n\}$) into different classes ($C = \{C_1, \ldots, C_k\}$) such a way that similar objects should be in the same class while dissimilars should be in different classes. These classes are called clusters. Clustering algorithms are used to determine the underlying structure of the object set. Often it is useful not to create statistical measures or perform tasks on the whole set of objects, but after a cluster analysing, doing it to the different clusters having similar objects which gives more accurate results. For example one can discover more proper connections between the features of cars if he/she first clusters the data and examines the cars with low top speed and low consumption (city cars) and cars with moderate top speed and very high consumption (luxury cars) separately, rather than handles all sorts of cars together.

*University of Szeged, Hungary e-mail: [koch, dombi]@inf.u-szeged.hu

This aim of clustering is usually described as maximizing the function [10]

$$Q_S^D(X,C) = Q_S(X,C) + Q^D(X,C)$$

where $Q_S(X,C)$ means the similarity between objects in the same cluster and $Q^D(X,C)$ the dissimilarity between the objects in different clusters. There are no exact mathematical formulas for $Q_S(X,C)$ or $Q^D(X,C)$ that could be acceptable for most of the cluster analysing tasks.

Dividing objects into two groups by minimizing the maximum distance in the clusters, can be done by bicoloring a maximum spanning tree in $O(n^2)$ steps. On the other hand dividing the objects into more than two clusters is an NP hard problem [1]. Although segmenting into more than two partitions can be done by sequentially dividing clusters into two, it often does not give optimal solutions and fails on very simple examples, for instance when we want to partition these objects into 3 groups (see Figure 1).

Figure 1: Problems with the multiple bipartition when dividing into two groups.

Detecting the number of clusters is also a very difficult problem, most of the clustering algorithms can only divide the objects into a number of clusters given by the user. Even there are special cases when appropriate clustering does not exist or the only good clustering is to order all of the objects into one cluster (e.g. integer coordinate pairs of the 2-dimensional space).

These are the main reasons why heuristical approaches are so popular among clustering techniques.

1.1 Two Main Types of Clustering Methods

Considering practical use there are two well-separable kinds of cluster analysing methods depending on the type of problems they have to solve.

- In the first group there are the faster algorithms having $O(n)$ complexity. These methods usually take the objects as points in the d-dimensional space (if the objects have d attributes) so we will refer to this group as coordinate-based methods. These methods can handle huge datasets with hundreds of thousands or even millions of records (e.g. calls of a telephone company, web log of an on-line store, shopping transactions of a supermarket, transfers of a bank,...etc.). Due to their quickness (note that clustering with these methods is faster than sorting the objects or finding the two closest/most similar objects) these algorithms give a rough segmentation and mostly recognize only spherical clusters. Using this group of clustering algorithms the user

usually has to give the number of clusters a priori. Most known representatives of this group are the Fuzzy C-Means [2], [11], [12] and the Kohonen Clustering Network [2].

- Methods in the second group have much lower speed, they have at least $O(n^2)$ complexity. These algorithms work on the distances/dissimilarities between the objects, which explains their time complexity. Since they are much slower than the coordinate-based ones, they are only good for smaller tables and most of these algorithms have to store the distance matrix, so their memory consumption can be quite large. The advantage of these methods is that they produce much better results. They may detect clusters with arbitrary shapes or sizes and determine the exact number of clusters. These algorithms are very closely related to shape recognition. For example they can recognize the arcs of a detached double spiral. The first distance-based methods were the agglomerative and divisive methods [1], [4] and one of the latest is HCS [5] which is also a graph theoretic approach but instead of using weighted edges HCS concentrates on edge-connectivity. One of the best algorithms in this group of methods is Chameleon, which was published in 1999 [7]. Chameleon has a very powerful recognizing capabilities, it detects clusters with arbitrary shapes and determines the number of clusters needed, still it has some serious drawbacks in practical use.

To emphasize the gap between the algorithms belonging to the two different groups let us show some calculations. Consider that a clustering algorithm segments 1000 objects in the 10-dimensional space in one second. If this algorithm belongs to the first group of methods, clustering 1 million objects takes approximately 17 minutes and with single precision real number representation it requires approximately 38 megabytes of memory while if the algorithm belongs to the distance-based group clustering 1 million records takes more than 1.5 weeks and the size of the distance matrix is 1.8 terabytes.

In this paper we will propose a new cluster analysing algorithm for the second group of methods. SmallSteps is a distance-based method and overcomes the difficulties of Chameleon while it keeps all the good features of it.

2 The Way Chameleon Works

Chameleon is one of the latest developed distance-based clustering method, which was published in 1999 by Karypis et al. [7]. It takes the objects as vertices of a graph with the weighted edges according to the k nearest neighbour graph on similarities between the objects. The weights of the edges are the similarity values. Chameleon has two main phases. In the first phase it creates small sub-clusters and merges them together into clusters in the second phase.

The sub-cluster creating part is done by a hypergraph-partitioning algorithm. Since partitioning a graph into a large number of equally sized subgraphs is an

Figure 2: Situation when the hypergraph-partitioning phase in Chameleon fails.

NP hard problem, Chameleon uses a heuristic technique called multilevel graph partitioning [8], [9].

To merge these sub-clusters Chameleon calculates special measures. The relative closeness is responsible to merge only clusters that have uniform density among the objects in the same cluster and relative inter-connectivity is for maintaining the similar inter-connectivity in the clusters.

Chameleon can work according to two different schemes.

- In the first scheme Chameleon introduces two technical parameters as thresholds. One for the relative closeness and one for the relative inter-connectivity. Pairs of clusters, whose calculated measures are above these thresholds, will be merged. Chameleon may terminate if there are no pairs of clusters whose relative closeness and relative inter-connectivity is above the thresholds or these parameters may be relaxed during the merging phase allowing Chameleon to create only one big cluster.

- According to the second scheme Chameleon uses a function to combine the relative closeness and relative inter-connectivity. This function is usually has the form

$$f(C_i, C_j) = RI(C_i, C_j) * RC(C_i, C_j)^{\alpha}$$

where $RI(C_i, C_j)$ and $RC(C_i, C_j)$ are the relative inter-connectivity and relative closeness between clusters C_i, C_j and α is a user specified parameter

to increase the importance of one of the two measures. After computing the goodness (f) of merging of all pairs of clusters, Chameleon combines the clusters with the best goodness value. Then the algorithm updates the $RI(C_i, C_j)$ and $RC(C_i, C_j)$ values and continues with the cluster pair selection. The result of this scheme is that we get one big cluster and the order of the mergings.

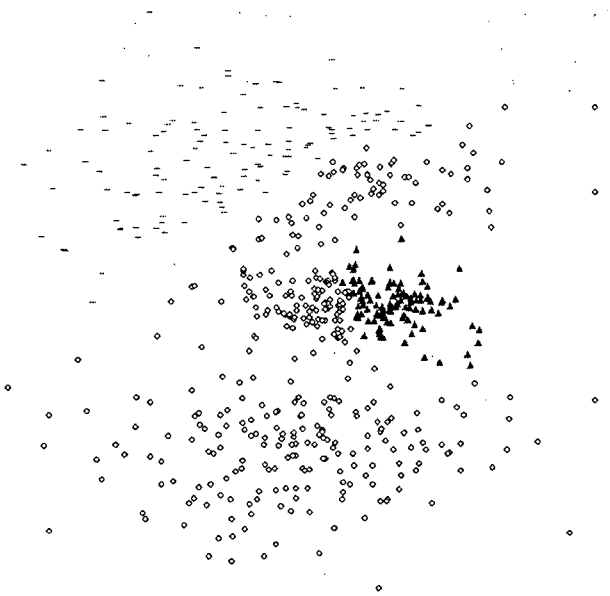


Figure 3: The result of Chameleon starting from wrong division, like on Figure 2.

The Chameleon method can handle nearly arbitrary shape of clusters and can detect quasi-automatically the number of clusters.

Experiments show that if the number of records is low, Chameleon works not well. The problem is that, with the heuristical graph partitioning algorithm, Chameleon at first divides the objects into a large number of relatively small subclusters which still have to be big enough to correctly compute their internal measures (e.g. the internal inter-connectivities which are used to compute the relative inter-connectivity between two clusters). If there are only small numbers of elements (i.e. less than 1000), very often one or more sub-clusters have intersections with more than one real cluster (see Figure 2). Since it has no error correction, this means that these clusters will be connected and the algorithm cannot separate them later (see Figure 3). Remember that methods in the distance-based group are best for relatively small tables.

On Figure 2 two sub-clusters (marked with 1 and 2) have common part with two genuine cluster. Cluster marked with 3 consists of objects of two separate groups

which indicates that the random part of the hypergraph-partitioning phase tried only wrong partitions. This last one occurs rather rarely, but plays major role in getting the wrong clustering of Figure 3.

Another problem with Chameleon is that it needs two technical parameters to detect the number of clusters, which are very hard to interpret by the user and in practical use it is a serious disadvantage.

Chameleon's graph partitioning algorithm is a non-deterministic procedure, which implies that the whole method is also non-deterministic.

Mainly these problems above inspired us to develop a new distance-based clustering method which is able to overcome these difficulties and gives a result of the same high quality as Chameleon does.

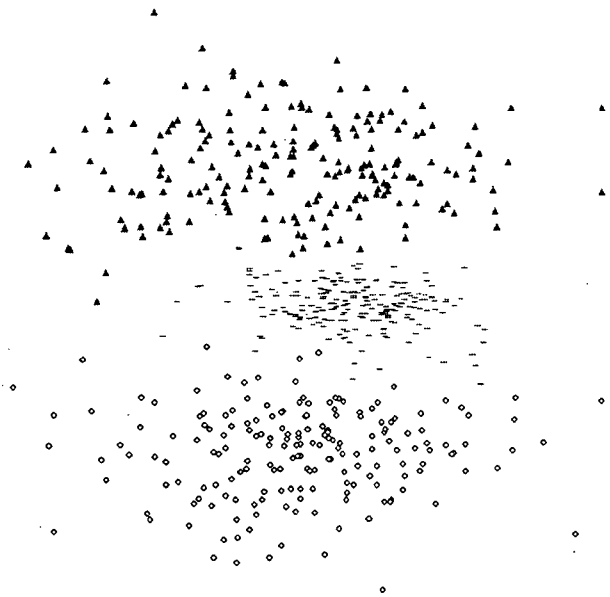


Figure 4: The result of SmallSteps on the object set of Figure 2 and Figure 3.

The developed SmallSteps

1. can be used for datasets of different size: from small to relatively large ones with up to 10000 records;

2. it needs no parameters, still it automatically detects the number of clusters and can recognize clusters with any shape and;

3. always gives the same result for a given set of objects.

3 SmallSteps: The New Procedure

3.1 The Algorithm of SmallSteps

The solution of SmallSteps, similarly to Chameleon's, comes from graph theory. The objects are considered as vertices of a graph and the distances between the objects are the weighted edges of the graph. The clusters are special connected graphs having edges with weights less than a cluster depended threshold called δ. These thresholds are recalculated in every iteration (see below) and unique to every cluster.

first phase	(1) order all elements into one cluster and calculate its δ (2) iterate $t = 1, \ldots, t_{max}$ (3) sort the clusters according to their δ (4) form the new clusters with the δ's (5) calculate the new δ's
second phase	(6) check whether merging clusters is possible
third phase	(7) iteratively process the oulier elements according to the selected strategy (8) check whether a new cluster can be form with the current outlier element (9) if not, order it to one of the existing clusters

Table 1: The algorithm of SmallSteps.

In the following part denote the number of clusters in the t^{th} iteration step by $|C^{(t)}|$. Denote the k^{th} cluster in the t^{th} iteration step by $C_k^{(t)}$ and denote its δ parameter by $\delta_k^{(t)}$ where $1 \leq k \leq |C^{(t)}|$. Finally denote the number of objects in cluster $C_k^{(t)}$ by $|C_k^{(t)}|$.

SmallSteps has three main phases.

- The first phase is the initial cluster-forming phase. At the beginning of this phase ($t = 0$) all the elements belong to one cluster ($C_k^{(t)}, k = 1$) and the initial δ ($\delta_k^{(t)}$) is calculated. Then in an iterative process the algorithm begins to search for special connected graphs with edge weights less than $\delta_k^{(t)}$ and the thresholds of the newly created clusters are recalculated. We repeat this process until a previously given number of iteration steps is reached (see Figure 5, Figure 6 and Figure 7). The δ's are responsible for creating clusters of upwardly bounded, nearly uniform densities, so it is very important to start every iteration with clusters having smaller δ's otherwise neighbouring clusters with smaller densities (higher δ's) would incorporate the denser ones.

The $\delta_k^{(t)}$ for the k^{th} cluster in the t^{th} iteration step is calculated based upon the average distance between the objects of the cluster $C_k^{(t)}$ and their closest neighbours from the same cluster.

$$\delta_k^{(i)} = \frac{\sum_{x_j \in C_k^{(t)}} f\left(\frac{1}{r}\sum_{l=0}^{r} d(x_j, x_{j(l)})\right)}{|C_k^{(t)}|}$$

where $x_{j(l)}$ is the l^{th} closest element to x_j and $d(x_j, x_{j(l)})$ is the distance/dissimilarity between x_j and $x_{j(l)}$. Here r is the degree of δ, i.e. the number of neighbours involved in the calculation and f is a monotone function which determines the way the average distance is taken into consideration. In Figure 4, 5, and 6 f was the same linear transformation.

Figure 5: The clusters found after the 1st iteration step in the first phase.

The degree of the δ's (r) controls the way SmallSteps works. If only one or a few neighbours are considered then the result of SmallSteps is very close to the result of a shape recognition algorithm.

Clusters are not always segmented into sub-clusters during this first phase, but merging is also possible still it is done very rarely.

The time complexity of this phase is $O(n^2 t_{max})$, where n is the number of objects, t_{max} is the number of cluster forming iterations and it is independent of the number of objects.

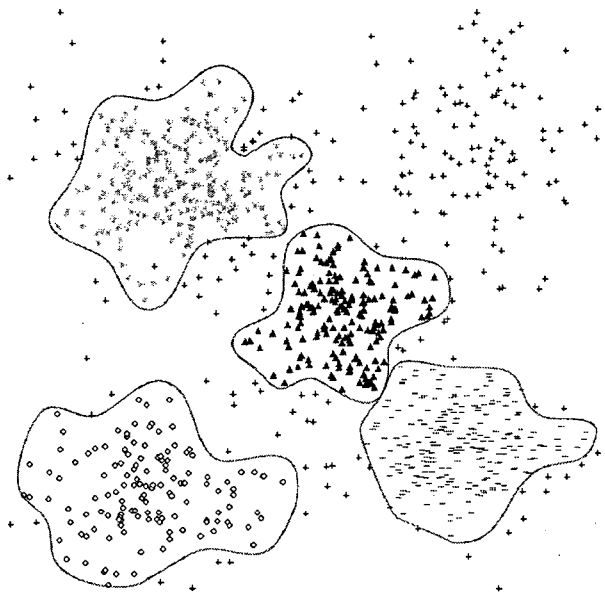


Figure 6: The clusters found after the 2nd iteration step in the first phase.

- The second phase is a merging phase. It makes decisions on merging some of the initially formed clusters based on their δ, size and distance. Although the first phase of SmallSteps is so powerful that usually there is no need to merge clusters in the second phase, our experiments showed that these merging calculations are worth to do.

  Merging objects has $O(|C^{(t_{max}+1)}|^2)$ time complexity, where $|C^{(t_{max}+1)}|$ is the number of clusters created during the first phase.

- The handling of outlier or noisy objects is done in the third phase (see Figure 8). Often datasets have outlier or noisy objects, which does not belong essentially to any clusters and could be left uncategorized, but in most cases the user wants to order all of the objects into clusters. If the user accepts outlier objects then this third phase should be skipped.

  Depending on the structure of outlier elements, in SmallSteps the following two operations can be done with them:

  - Let them form new clusters.
    If there is a group of outlier objects that are far enough from the existing clusters and have enough elements which are close to each other to form a new connected subgraph with edge weights less than their special δ then these objects are allowed to create a new cluster.
  - Order them to existing clusters.

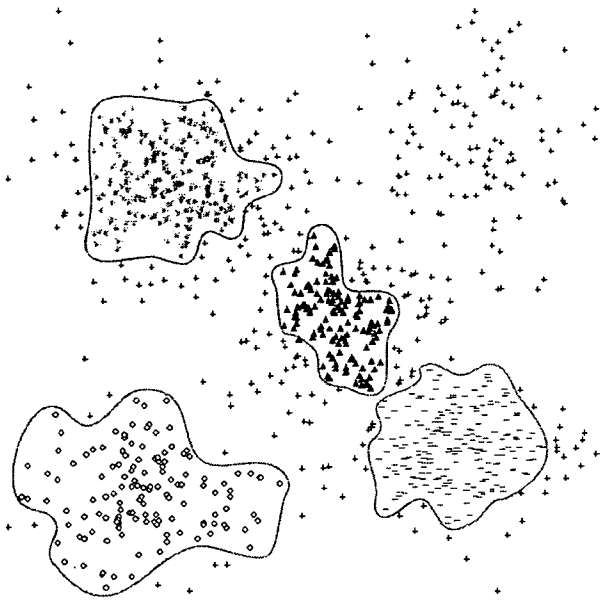


Figure 7: The clusters found after the 5^{th} iteration step in the first phase.

The order of outlier objects influence the result so we have developed different strategies:

- The simplest is to choose always those outlier objects that are the nearest to an existing cluster.
- We may follow a postponing strategy and choose those objects first whose classification are the easiest, i.e. which are close to a cluster but far from the other clusters.
- According to the BestFit method those outlier objects are always classified first whose distance to a cluster best fit to the δ of the cluster. This method tries to maintain the uniform density of the clusters.

The time complexity of the last phase is $O\left(n_{\text{outlier}} * \max\{n_{\text{outlier}}, n_{\text{classified}}\}\right)$, where n_{outlier} and $n_{\text{classified}}$ are the numbers of outlier and classified elements at the beginning of the third phase, respectively.

The overall time complexity of SmallSteps is

$$O\left(n^2 t_{max} + |C^{(t_{max}+1)}|^2 + n_{\text{outlier}} * \max\{n_{\text{outlier}}, n_{\text{classified}}\}\right) = O(n^2).$$

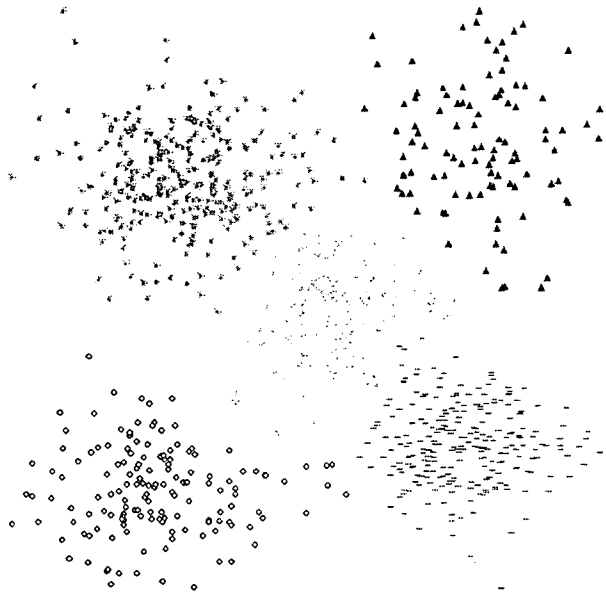


Figure 8: The clusters found after the outlier handling phase.

3.2 Handling Bounding Objects

If two clusters are in touch by only a few objects then by searching for connected graphs the algorithm will find that these clusters could form one cluster since there is a path between the elements of the two clusters with edges less than the δ's of the clusters. To avoid such aggregation it is useful to find the boundary objects of the clusters and if two clusters are connected with only bounding objects then the algorithm should not merge these clusters. The algorithm can recognize the bounding objects by counting their intra-cluster neighbourhood which is the number of objects from the given cluster that are closer to the object than the δ of the cluster. The bounding objects have fewer neighbouring objects than the inner ones.

3.3 Inner Analysis of a Cluster

SmallSteps provides a useful additional feature. If the user wants to analyse a cluster in more detail, he/she can specialize this cluster by segmenting it into sub-clusters. The segmentation is done by iteratively decreasing the original δ of the cluster and searching for connected graphs with edge weights less than its new δ. The specializing procedure terminates when the algorithm is able to segment the cluster into sub-clusters of acceptable size or when it turns out that such a segmentation is not possible (see Figure 9).

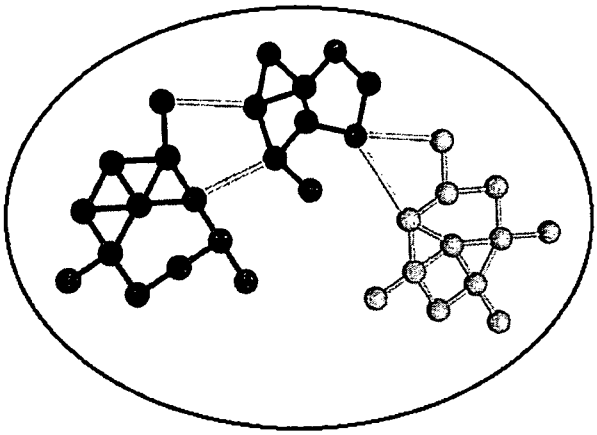


Figure 9: Segmenting a cluster into sub-clusters.

4 Example

On Figure 5, Figure 6, Figure 7 and Figure 8 the different stages of SmallSteps is shown during clustering.

Figure 5 shows the situation after the first iteration step of the first phase. The beginning one big cluster fell into 3 sub-clusters framed with thin lines. The objects marked with "+" signs are considered as outlier objects in this step. Note that one of the sub-clusters, having objects marked with "-" signs, still contains two genuine cluster.

After the second iteration step of the first phase (see Figure 6) the wrongly merged clusters broke up and the final clusters began to take form.

The following iteration steps in the first phase only refine the borders of clusters found in the second step. In this example after the 8^{th} step these refinements stop and no change is made in the subsequent iteration steps. On Figure 7 the division after the 5^{th} iteration step is shown.

The core of four of the five genuine cluster is detected in the first phase and there is no need to merge clusters in the second phase.

The detection of the last cluster is done in the third phase during the outlier handling process (see Figure 8). In this phase the algorithm detected the possibility of forming a new cluster from the outlier objects and the objects not involved in this cluster were incorporated into one of the existing cluster.

5 Results

We tested SmallSteps on numerous sample databases, four of them can be seen on Figure 8, Figure 10, Figure 11 and Figure 4.

The test shown on Figure 8 was performed on 1000 objects from 5 clusters taken from 5 normal distribution. The clusters have different densities with 100 to 300 objects.

The test of Figure 10 contained 1024 objects in 6 clusters of highly different densities. The two clusters on the bottom of Figure 10 are dense clusters connected with a rarer zone, but this zone is still denser than the other clusters.

On Figure 11 the result of a test on 2000 objects is shown. The nearly all of the clusters are only vertical lines with outlier objects on the endings. The objects were taken from the abalone database from the Repository of Machine Learning Databases and Domain Theories maintained by the University of California at Irvine.

The sample database of Figure 4 contained 600 objects taken from 3 normal distribution, all of them having 200 objects. It is a very difficult problem because of the high noise and the elliptical clusters. We also tested 3 coordinate based methods on the sample sets and none of them could solve this problem adequately.

In Table 2 the running times on the different tests are shown. The values are in milliseconds and measured on a 550MHz Intel Pentium III machine with 128MB RAM. Each test was performed 20 times.

In the first column the measured time of the full SmallSteps algorithm are shown while in the second column we skipped the outlier handling phase. We implemented Chameleon and the running times of the implementation is shown in the third column. Testing Chameleon is done by creating 30 sub-clusters with the hypergraph-partitioning algorithm and merging them into the given number of clusters according to the second scheme.

Test	SmallSteps	SmallSteps without 3^{rd} phase	Chameleon
Figure 4 (avg)	152.023	142.467	422.554
Figure 4 (σ)	1.682	5.532	9.530
Figure 8 (avg)	416.872	367.205	984.818
Figure 8 (σ)	0.985	4.772	16.762
Figure 10 (avg)	356.773	338.504	1050.196
Figure 10 (σ)	1.619	0.941	27.605

Table 2: Test results of SmallSteps and Chameleon.

6 Summary

In SmallSteps, the number of clusters evolves automatically during the three phases (mainly during the first phase) hence no user interaction is needed for giving this number. Instead of relative inter-connectivity and relative closeness, SmallSteps calculates δ's to perform cluster forming. These δ's can be computed from only a few objects, which means that SmallSteps works well on smaller tables too.

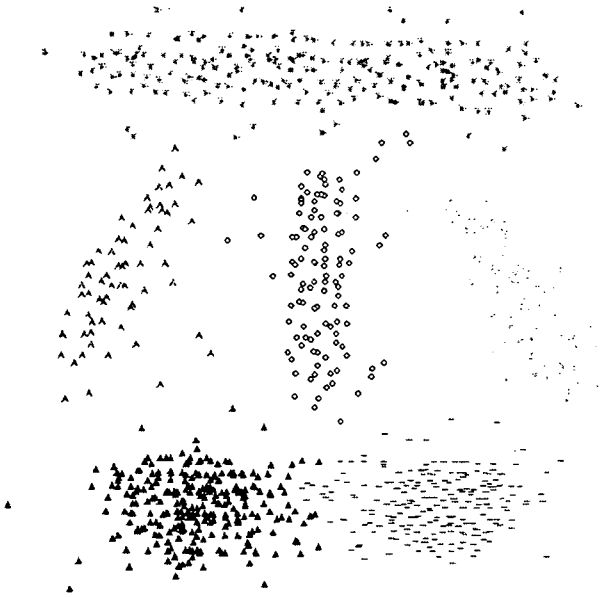


Figure 10: Detecting elliptical clusters with different densities and sizes.

SmallSteps can recognize clusters with different densities (see Figure 10) because every cluster has its own δ and since SmallSteps searches for connected graphs it can recognize clusters with arbitrary shapes or sizes (see Figure 11).

The algorithm of SmallSteps is deterministic so it will give always the same output for a given set of objects.

Since the outlier or noisy objects are handled in the last phase and till then they are eliminated from the processing by the first few iteration steps of the first phase SmallSteps is not too sensible to this kind of objects (see Figure 4).

While Chameleon is a kind of greedy, agglomerative clustering procedure and never corrects the errors made during its merging process, SmallSteps rather resembles to a divisive clustering method, but it has merging steps too and it has some error-correcting feature in all the three phases. Practical experiments showed that, among hierarchical algorithms without error correction, the divisive methods usually outperformed the agglomerative ones because divisive methods needed fewer steps to create the segmentation [4].

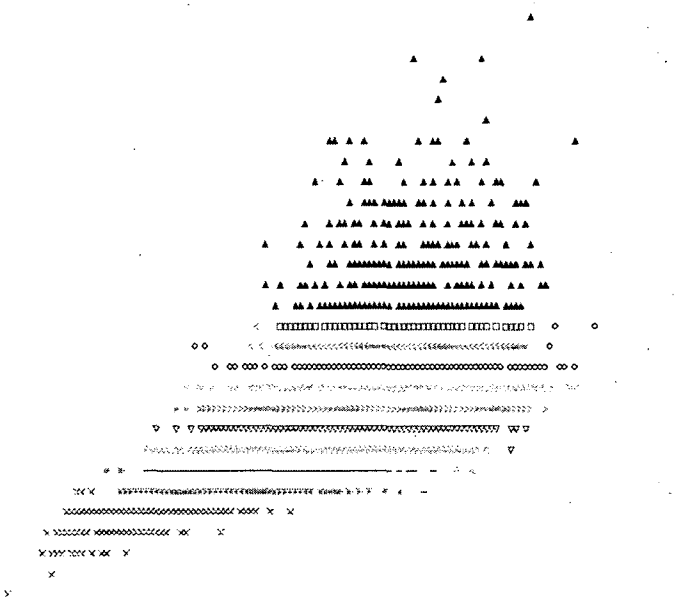


Figure 11: SmallSteps is very close to shape detection.

References

[1] Charles J. Alpert and Andrew B. Kahng: *Splitting An Ordering into a Partition to Minimize Diameter*, Journal of Classification, 14 (1997) 51-74.

[2] James C. Bezdek, Erik Chen-Kuo Tsao and Nikhil R. Pal: *Fuzzy Kohonen Clustering Networks*, Pattern Recognition, 27 (1994) 757-764.

[3] Rajesh N. Davé and Raghu Krishnapuram: *Robust Clustering Methods: A Unified View* IEEE Transactions on Fuzzy Systems, 5 (1997) 270-293.

[4] A. Guénoche, P. Hansen and B. Jaumard: *Efficient Algorithms for Divisive Hierarchical Clustering with the Diameter Criterion*, Journal of Classification, 8 (1991) 5-30.

[5] Erez Hartuv and Ron Shamir: *A clustering algorithm based on graph connectivity* Information Processing Letters, 76 (2000) 175-181.

[6] A. K. Jain, M. N. Murty and P. J. Flynn: *Data Clusterin: A Review* ACM Computing Surveys, 31 (1999) 264-323.

[7] George Karypis, Eui-Hong Han and Vipin Kumar: *Chameleon: Hierarchical Clustering Using Dynamic Modelling*, IEEE Computer, 32 (1999) 68-75.

[8] George Karypis and Vipin Kumar: *Analysis of Multilevel Graph Partitioning*, Technical Report TR 95-037, University of Minnesota, Department of Computer Science, 1995.

[9] George Karypis and Vipin Kumar: *Multilevel k-Way Partitioning Scheme for Irregular Graphs*, Journal of Parallel and Distributed Computing, 48 (1998) 96-129.

[10] Jan W. Owsiński: *Clustering - modelling, capabilities, limits, applications*, Control and Cybernetics, 24 (1995).

[11] Mika Sato, Yoshiharu Sato and Lakhmi C. Jain: *Fuzzy Clustering Models and Applications*, Physica-Verlag Heidelberg, 1997, ISBN 3-7908-1026-6.

[12] Ching-Chang Wong and Chia-Chong Chen: *A Clustering Based Method for Fuzzy Modeling*, IEICE Transactions on Information and Systems, E82-D (1999) 1058-1065.

Acta Cybernetica 15 (2001) 257–278.

The Debug Slicing of Logic Programs*

Gyöngyi Szilágyi†, László Harmath† and Tibor Gyimóthy†

Abstract

This paper extends the scope and optimality of previous algorithmic debugging techniques of Prolog programs using slicing techniques. We provide a dynamic slicing algorithm (called Debug slice) which augments the data flow analysis with control-flow dependences in order to identify the source of a bug included in a program.
We developed a tool for debugging Prolog programs which also handles the specific programming techniques (cut, if-then, OR). This approach combines the Debug slice with Shapiro's algorithmic debugging technique.

1 Introduction

Slicing methods are widely used for the debugging [25], testing [2] and maintenance of imperative programs [1, 12]. Intuitively, a slice should contain all those parts of a program that may affect the variables in a set V at a program point p [26]. Slicing algorithms can be classified according to whether they only use statically available information (*static slicing*), or compute those program points which influence the value of a variable occurence for a specific program input (*dynamic slice*). Dynamic slicing methods are more appropriate for debugging than static ones as during debugging we generally investigate the program behaviour under a specific test case. The main advantage of using a dynamic slice during debugging is that many statements can be ignored in the process of bug localization.

Different dynamic slicing methods have been introduced for debugging imperative programs [23]. Most of these methods are based on a dependence graph which contains the explicit control dependences and data dependences of the program. In [9, 14] a slicing method was introduced for logic programs, and this method being used to improve the efficiency of the Shapiro's algorithmic debugging algorithm [19]. The slice presented in [9] contains those parts of a program that actually have an influence on the value of an argument of a predicate. This type of slice (called *data flow slice*) is safe if the structure of the proof tree for a goal is not changed [22].

*This work was supported by the grants OTKA T52721, and IKTA8/99.

†Research Group on Artificial Intelligence Hungarian Academy of Sciences Address: 6720 Hungary, Szeged, Aradi vértanuk tere 1. E-mail:{szilagyi, harmat, gyimi}@inf.u-szeged.hu

However, during debugging to find a source of a bug (i.e. a bug instance) we also need to identify those predicates that actually did not affect an argument in a predicate but could have affected it had they been evaluated differently (had their boolean outcome been different). We can say that these predicates are in the Potentially Dependent Predicate Set. Note that a different evaluation of the predicates in this set could change the success branch of the SLD-tree (where the bug was manifested).

Consider the following example.

Example 1 *The buggy program is:*

1. p(A,X) :- q(A,X).
2. q(A,X) :- **A > 0**, *X is 2.*
3. q(A,X) :- X is 3.

The correct program should be:

1. p(A,X) :- q(A,X).
2. q(A,X) :- **A = 0**, *X is 2.*
3. q(A,X) :- X is 3.

Executing this program for the goal $p(0, X)$ the given solution is $X = 3$, while we expect $X = 2$. So a bug must be included in the program somewhere.
Creating the dynamic data flow slice for an instance of X, it does not contain the buggy predicate $A > 0$ because X does not exactly depend on the predicates of clause 2, there being only control dependences between them. This means that if $A > 0$ had been evaluated differently it could have affected the solution of X. Our new slicing approach contains the buggy predicate $A > 0$ (see Section 4.2).

In this paper we introduce a new type of slicing called *Debug slicing* for Prolog programs without side effects. A Debug slice of an Augmented SLD-tree includes those predicates that may affect the value of an argument in any success branch's predicate. So this slice is very suitable for debugging. The Debug slice is the set of predicates which contains the Potentially Dependent Predicates and their data dependences.

This slicing method has been integrated into an interactive algorithmic debugging tool to reduce the number of questions to the user during a debugging session [14]. The size of the debug slice is larger than the size of the data flow slice, but the data flow slice is not safe for debugging. On the other hand the Debug slice contains all parts of the program that may be responsible for the incorrect behaviour at some selected argument position.

In the next section the basic concepts of logic programming, algorithmic debugging and slicing are presented. Section 3 then provides a detailed description of the construction of those structures needed in an outline of the Debug slice algorithm (Augmented SLD tree, Skeleton(n), (Directed) Proof Tree Dependence Graph, General Data Flow Slice). The computation of the Debug slice on the basis

of these structures is described in Section 4. The first results of a prototype implementation of Debug slice algorithm are discussed in Section 5. Finally, in Section 6 we summarize related work and outline further studies.

2 Preliminaries

In this section we present some basic concepts (logic programming, algorithmic debugging, slicing) needed to outline the Debug slicing algorithm.

2.1 Logic Programming

A **first order alphabet** consist of variables, predicate symbols and function symbols (which include constants) [24].

A **variable** is represented by an upper case letter followed by a string of lower case letters and/or digits.

A **function symbol** is a lower case letter followed by a string of lower case letters and/or digits.

The **constants** include integers and atoms, a constant is a function symbol of arity 0. The symbol for an **atom** can be any sequence of characters.

A variable is a **term**, and a function symbol followed by bracketed n-tuple of terms is a (compound) term. Thus $f(g(X), head)$ is a term when f, g and *head* are function symbols and X is a variable.

A predicate symbol immediately followed by a bracketed n-tuple of terms is called an **atomic formula**, or atom.

Let $h, a_1, \cdots, a_m$ be atomic formulae for some $m \geq 0$ and let $X_1, \cdots, X_l$ be all variables occurring in these formulae.

Then the formula $\forall X_1 \cdots \forall X_l(h \leftarrow a_1, \cdots, a_m)$ is called a **definite clause**. If $m = 0$ the formula is called a **fact**. The atomic formula h is called the *head* of the clause, while $a_1, \cdots, a_m$ is called its *body*. A **goal** is a definite clause with empty head. Since all variables of a definite clause are universally quantified we can omit the quantifiers.

A clause or an atom is **ground** if it has no variable.

A **normal program** is a set of program clauses.

A **substitution** is a finite set (possible empty) of pairs of the form $X \rightarrow t$, where X is a variable and t is a term and all the variables X are distinct. For any substitution $\sigma = \{X_1 \rightarrow t_1, \cdots, X_n \rightarrow t_n\}$ and term s, the term $s\sigma$ denotes the result of replacing each occurrence of the variable X_i by t_i $(i = 1, \cdots, n)$. The term $s\sigma$ is called an **instance** of s.

A substitution σ is called a **unifier** for two terms s_1 and s_2 if $s_1\sigma = s_2\sigma$. Such a substitution is called **the most general unifier** of s_1 and s_2 if for any other unifier σ_1 of s_1 and s_2, $s_1\sigma_1$ is an instance of $s_1\sigma$. If two terms are unifiable then they have uniquie most general unifier.

A **computation of a logic program** P [19] can be described informally as follows. The computation starts from some initial goal g and can have two results:

success or failure. If a computation succeeds, then the final values of the variables in g are considered of as the output of the computation. A given goal can have several successful computations, each resulting in a different output.

The computation progresses via nondeterministic goal reduction, at each step we have some current goal $G = g_1, \cdots, g_n$. A clause $C = a \leftarrow b_1, \cdots, b_k$ in P is then chosen nondeterministically; the head of the clause a is then unified with g_1, say, with substitution σ, and the reduced goal is $G' = (b_1, \cdots, b_k, g_2, \cdots, g_n)\sigma$. The computation terminates when the current goal is empty. Then G' is said to be *derived* from G and C.

Let P be a logic program and G a goal. A *derivation* of G from P is a possible infinite sequence of triples $< G_i, C_i, \sigma_i >$, $i = 0, 1, \cdots$ such that G_i is a goal, C_i is a clause in P with new variable symbols not occuring previously in the derivation, σ_i is a substitution, $G_0 = G$, and G_{i+1} is derived from G_i and C_i with substitution σ_i, for $i \geq 0$. If there is a derivation of G from P such that $G_l = \Diamond$ (the empty goal) for some $l \geq 0$ we say that P *succeeds* on G. We assume by convention that in such a case $C_l = \Diamond$ and $\sigma_l = \{\}$.

2.2 Algorithmic Debugging of Logic Programs

Algorithmic debugging is a process where the user and a debugging system interactively try to locate the source of an externally visible bug in the program [19]. There are a number of features of Prolog program which distinguish it from other programming languages. A Prolog program can have both a declarative and a procedural reading, and may or may not be multi-directional and even it can be nondeterminate. The computation model of Prolog is based on goal invocation as well as goal success and failure. Thus errors in Prolog programs occur when, for example, they finitely fail on goals that should succeed, or succeed on goals that should fail.

A *bug manifestation* is undesired program behaviour, i.e. an undesired sequence of solutions computed by the program for a goal. A *bug instance*, which is a predicate instance, is a cause for a top goal bug manifestation. Our algorithm identifies a set of predicates of a program which can cause the bug mainfestation (i.e. an undesired solution with respect to a variable of the top goal).

Shapiro's algorithm [19] traverses the proof tree of a program in different ways and asks the user about the expected behaviour of each resolved goal. The *bottom-up method* traverses the proof tree in postorder manner and asks the oracle about the correctness of the computed values of the nodes. If the result at a node is incorrect and all sons of this node are evaluated correctly the algorithm identifies the clause applied to this node as a buggy one. The query complexity of this method is linear in the size of the tree.

The second method investigates the nodes in a *top-down* manner. If the result computed at a node is evaluated correctly by the oracle then the algorithm does not visit the nodes inside the sub-tree. Using this approach the query complexity can be reduced to a linear dependence in the depth of the proof tree.

The most efficient technique is the *divide-and-query* startegy which requires a num-

ber of queries logarithmic in the size of the proof tree. The divide-and-query algorithm splits the proof tree into two approximately equal parts, and makes a query for the node at the splitting point. If this node gives an incorrect evaluation the algorithm goes on recursively to the sub-tree associated with this node. If the node's answer is correct its sub-tree is removed from the tree and a new mid-point is computed.

A Prolog program may use a number of programming techniques specific to Prolog (cut, if-then, OR). We developed a tool for debugging Prolog programs incorporating these specific programming techniques as well for finding the source of a bug, i.e. for identifing a bug instance.

2.3 Slicing

Slicing is a program analysis technique originally developed for imperative languages [26]. Later improvements are presented in [23, 21, 15, 13].
Intuitively a program slice with respect to a specific variable V at some program point p (which can be a variable or an argument position of a predicate) contains all those program points that may affect the value of the variable or may be affected by the value of the variable. The tuple $< V, p >$ is called a *slicing criterion* and a slice is computed with respect to one.
Slicing techniques can also be classified into *static* and *dynamic* ones.
Static slicing is based on an analysis of the program without executing it so it may be imprecise if it contains data flow which is actually not manifested during a particular execution.
Dynamic slicing is based on the program's execution and hence extracts the precise data flow. A dynamic slice may be different for each execution and so shall always be produced separately whenever the program run.
In addition, slicing can be classified into *forward* and *backward* types. Suppose our slicing criterion is $< V, p >$. Forward slicing with respect to $< V, p >$ contains all those program points that may have its value modified if $< V, p >$ is modified. Backward slicing contains all those program points which, if modified, might change the value of V.
In the case of imperative languages a possible program representation for the program's dependences is the Program Dependence Graph [7, 16, 11].

The problem of slicing logic programs is more complicated than for the imperative case. Before a slice over a logic program can be produced, its implicit data flow has to be approximated. To approximate the data flow the implicit input/output data dependences have to be extracted from the program.
Program slicing has been widely studied for imperative programs [23], but research on slicing logic programs is just beginning. To our knowledge, only a few papers deal with the problem of slicing logic programs [9, 20, 27, 22].
The main contribution of this article was to furnish a dynamic slicing algorithm which augments the data flow analysis with control-flow dependences so as to make the slicing algorithm better suited for debugging.

3 Basic structures and theorems for constructing the Debug Slice

Our slicing is based on a dependence based approach. In [9] a Dependence Graph was constructed for a proof tree i.e. for a success branch of the SLD-tree. We would also like to extend this definition to the failure branches of the SLD tree. This is why this section provides a detailed description of the necessary structures needed to outline this extension: the Augmented SLD-tree, Skeleton(n) (a modified derivation tree), (Directed) Proof Tree Dependence Graph (PTDG) and General data flow slice. The computation of the Debug slice on the basis of these structures is described in the next section.

The *Augmenetd SLD-tree* shows the execution order of the statements for a given input.

Skeleton(n) is always built for one branch of the Augmented SLD-tree, and its nodes represent the data flow information needed for preparing the *Proof Tree Dependence Graph.*

A *General slice* of a logic program with respect to a variable V is constructed using the Proof Tree Dependence Graph, which contains those predicates of a derivation that may affect the value of V.

3.1 The Augmented SLD-tree of Logic Programs

The derivation of a goal from a program P can be represented by a tree called SLD-tree. Each branch of the SLD-tree [17] is a derivation of a program for a goal. Branches corresponding to successful derivations are called *success branches*, while branches of the infinite derivations are called *infinite branches*; those corresponding to failed derivations are called *failure branches.* The Prolog interpreter searches the SLD-tree to find success branches. The Prolog system always selects the leftmost atom in a goal along with a depth-first search rule. The program clauses are then tested in their original order in the program.

An SLD-tree may have many failed branches and very few or just one success branch. Control information supplied by the user may prevent the interpreter from construction of failed branches. To control the search the concept of *cut*(!) is introduced in Prolog. The atom "!" is handled as an ordinary atom in the body of a clause. When a *cut* is selected for resolution it succeeds immediately (with the empty substitution) [18]. The node where *cut* is selected will be called the *cut node.* A cut node may be reached again during backtracking. In this case the normal order of tree traversal is altered - by definition of *cut* the backtracking continues above the node origin (!). (If *cut* occurs in the initial goal, the execution simply terminates). So *cut* has the following effect: after success of "!" no backtracking to the literals in the left-hand part is possible. However, in the right-hand part execution goes on as usual.

We add these pieces of information to the SLD-tree, identify each node with an unique mark, and use a list ($pred\_def\_ref()$) in order to know which program

clauses (corresponding to the selected predicate) are used at a node to execute the next step. We also deal with the pruning effect of cuts. The following definition provides a formal description of the modified SLD-tree. The node label contains the whole list of goals $((G', R) = (a_1, a_2, \cdots, a_n))$. The actual goal (G') is the first in this list (a_1 in our case). A node has a child for every program clause whose head (h_m) could be unified with the actual goal. The list of these clauses for every node is given in *pred_def_ref()* (Definition 1.1). If the actual goal were *cut(!)*, the corresponding branches of the tree would be pruned (Definition 1.2 and 1.3).

Definition 1 *Let P be a Prolog program and G a goal. An* **augmented SLD-tree** *for $P \cup \{G\}$ is a tree which satisfies the following:*

- Each node label is triple $< Mark, (G', R), pred\_def\_ref(G') >$, where $Mark$ is a *unique identification* of the node, (G', R) is a (possibly empty) conjunction of goals (*resolvent*). G' is the leftmost goal in the resolvent (called *selected goal*) and $pred\_def\_ref(G')$ is a *reference list* for the predicate definitions corresponding to the leftmost goal G'. We assign the empty resolvents with a true value.

- The root node is $< Mark, (G, true), pred\_def\_ref(G) >$.

- Let $< M, (a_1, a_2, \cdots, a_k), pred\_def\_ref(i_1, \cdots, i_l) >$ be a node in the tree (so a_1 is the selected atom), where $i_m (m = 1, \cdots, l)$ is the identity number of input clauses $h_m \leftarrow b_{m_1}, \cdots, b_{m_q}$ such that a_1 and h_m are unifiable with most general unifier σ.

  1. Then this node has a child
     $< M_{i_m}, (b_{m_1}, b_{m_2}, \cdots, b_{m_q}, a_2, \cdots, a_k)\sigma, pred\_def\_ref(b_{m_1}) >$ for each $i_m (m = 1, \cdots, l)$. The edges immediately below a node and also the $pred\_def\_ref()$ list are ordered from left to right, according to the program clause order.

  2. If $h_m \leftarrow b_{m_1}, \cdots, b_{m_q}$ has cuts the child is
     $< M_{i_m}, (b'_{m_1}, b'_{m_2}, \cdots, b'_{m_q}, a_2, \cdots, a_k)\sigma, pred\_def\_ref(b'_{m_1}) >$,
     where $b'_{m_1}, \cdots, b'_{m_q}$ are obtained from $b_{m_1}, \cdots, b_{m_q}$, replacing all cuts with one same unique annotation such as "cut(M)", where M is the node's identification mark.

  3. If the following selected atom (b'_{m_1} or a_2) were cut(M), we use the pruning rule below, and the next element of the list that follows cut(M) will be the selected goal.
     The pruning rule: If "Mark" is the argument of cut(), consider the path W from the actual node up to the node marked Mark. All descendants of this node to the right of W are removed.

- Nodes with an empty R list in resolvent have no children.

We will refer to the Augmented SLD-tree simple as SLD-tree. During the execution of a program for a goal to find the first success branch of the SLD-tree, only a part of it is walked by the Prolog interpreter. We will call this part of the SLD-tree the **Trace-tree** because we can build the Trace-tree from the trace of a program for a goal given by the interpreter. Figure 1 shows the Trace-tree of the program in Example 2 for the goal $a(Y)$.

Example 2 *We now illustrate our definition of the Trace-tree in a simple example:*
1. a(Y):-b(X), c(Y), d(X).
2. c(Y):-b(Y), e(Y).
3. b(1).
4. b(2).
5. b(3).
6. d(2).
7. e(3).

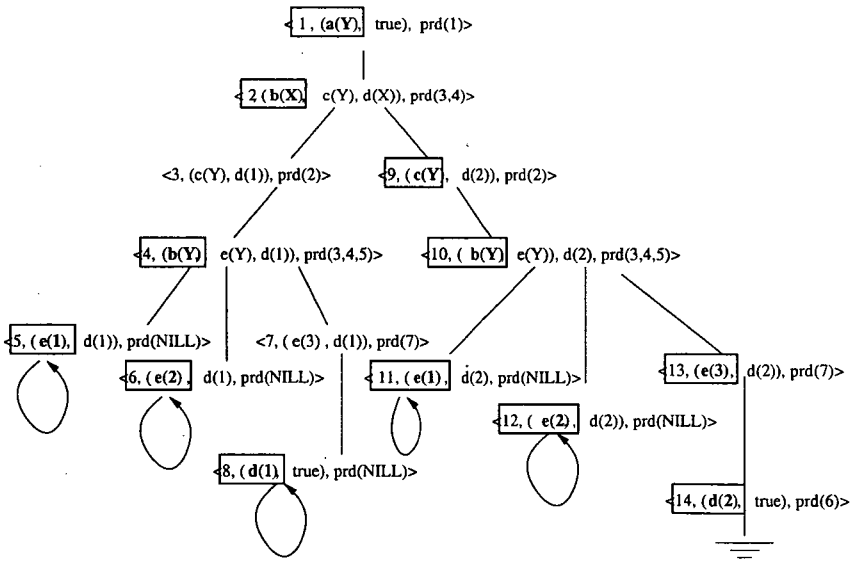


Figure 1: The Trace-tree for the goal a(Y) and the Debug slice in frames (see Section 4.2).

Example 3 *Figure 2 shows the pruning effect of the cuts in the following example for the goal $a(X)$.*
1.a(X) :- b(X), c(X).
2. a(X) :- g(X).
3. b(X) :- c(X), !, d(X).
4. b(X) :- e(X), h(X).
5. c(1).

6. c(2).
7. d(2).
8. e(1).
9. h(1).
10.g(3).

The removed part of the Trace-tree is depicted by a broken line (the pruning effect of the cut).

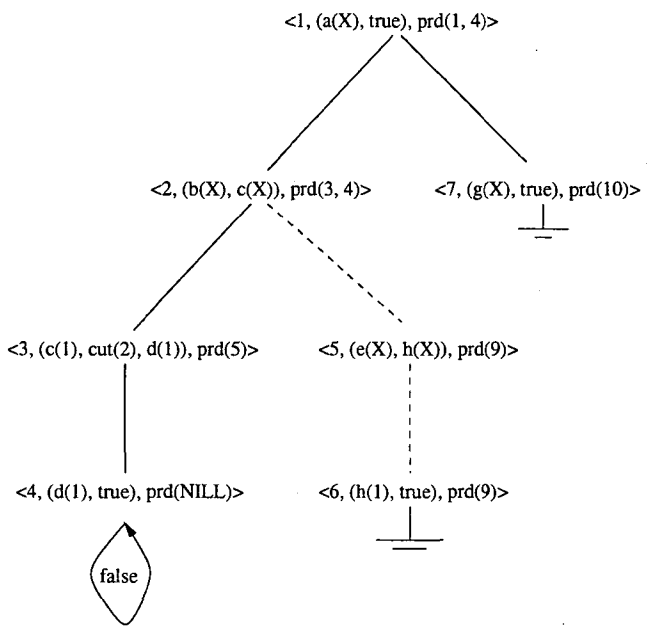


Figure 2: The pruning effect of the cut in Example 3 for the goal a(X).

3.2 Skeleton(n)

The SLD-tree representation is unsuitable for representing the data flow information of a logic program (for a given goal). The structure $Skeleton(n)$ [24] is used to represent this information, where n identifies a leaf node of the SLD-tree. $Skeleton(n)$ is basically a derivation tree defined for one branch of the SLD-tree (from the root to the node marked by n). To improve the approximation of the implicit data flow $Skeleton(n)$ contains directionality information as well.

We will use the notion of *clause instance* ($c\sigma$) which means that a substitution σ is applied for every predicate of c.

We extended the definition of the derivation tree used in [24] to our case.

Definition 2 Proof Tree
For a program P a **proof tree** *is any labeled, ordered tree T such that*

1. *Every node is labeled by an instance name of a clause of P.*
2. *Let n be a node labeled by an instance name $< c, \sigma >$, where c is the clause $h \leftarrow a_1, \cdots, a_m$ $(m \geq 0)$ and σ is a substitution. Then n has m children, for $i = 1, \cdots, m$, the i-th child of n being labeled $< c_i', \sigma_i' >$, where c_i' is a clause with head h_i' such that $a_i\sigma = h_i'\sigma_i'$.*

The reasoning behind this definition is that every tree is obtained by combining appropriate instances of program clauses. The precise meaning of "appropiate instances" is expressed in condition 2. A logic program defines a set of derivation trees. This may be viewed as a semantic of definite programs, and can be related to the concepts of proof defined in symbolic logic.
A derivation tree represents one branch of the SLD-tree (one derivation), but in a more suitable format for representing the data flow.
Let us modify this definition to suit our present needs.
We need not know exact the substitution itself, it is enough to know which variables are ground at call of a predicate and which are ground at success. Basically having to investigate directionality information of the tree using some groundness annotation.
Let us suppose that we can identify each argument position of the clauses of a derivation tree with a tuple (the formal definition of the argument position is given at the end of this subsection).

Groundness information associated with a derivation tree will be expressed as an annotation of its argument positions. The annotation classifies the argument positions of a derivation tree. The positions are classified as *inherited* (marked with $\downarrow$), *synthesized* ($\uparrow$) and *dual* ($\updownarrow$). An annotation is *partial* if some positions are dual. Formally speaking, an annotation is a mapping ν from the positions in the set $\{\downarrow, \uparrow, \updownarrow\}$ [6].

The intended meaning of the annotation is the following. An **inherited argument position** is a position in which every variable is ground at time of calling, that is when the equation involving this position is first created during the construction of the derivation tree. A **synthesized argument position** is a position in which none of the variables are ground at time of calling, and every variable is ground at success, that is when the subtree having the position in its root label is completed in the computation process. The **dual argument positions** of a proof are those which are neither inherited nor synthesized. The annotations are collected during the execution of a program for a given goal. We notice that the argument positions are annotated at the present version of our tool. But the annotation of the variable positions would provide more precise dependences so we are planning to extend the annotation to variable positions.
We now introduce the following auxiliary terminology relevant to the annotated positions of a LP program. The inherited positions of the head atoms and the synthesized positions of the body atoms are called **input positions**. Similarly, the

synthesized positions of the head atoms and inherited positions of the body atoms are called **output positions**. The others are **dual**. Note that dual positions are not strictly classified as input or output ones. Alternatively, if we say that a position is annotated as an output we mean that it is annotated as inherited provided it is a position in a body atom, or annotated as synthesized if it is a position of the head of a clause.

Now we are ready to define $Skeleton(n)$.

Definition 3 Skeleton(n)
Let T be a SLD-tree, $< n, (G', R), pred\_def\_ref(G') > \in nodes(T)$ a leaf node identified by n. Consider the path W in T from the root to n, which identifies a derivation for the root goal.
Then, $Skeleton(n)$ is a labeled ordered tree such that

1. *Every node is labeled by a double $< Mark, \nu(c) >$, where Mark is an unique identification, and $\nu(c)$ is the annotated clause instance.*

2. *The root node is labeled by $< 1, \nu(c) >$, where the root goal was unified with c during the given derivation.*

3. *Let k be a node labeled by*
   $< Mark, p([X_1, \nu(X_1)], \cdots, [X_{k_0}, \nu(X_{k_0}])) :-$
   $a_1([Y_1, \nu(Y_1)], \cdots, [Y_{k_1}, \nu(Y_{k_1})]), \cdots, a_m([V_1, \nu(V_1)], \cdots, [V_{k_m}, \nu(V_{k_m}])) >$.
   Then k has m children, for $i = 1, \cdots, m$, the i-th child of k being labeled $< Mark, \nu(c'_i) >$, where c'_i is a clause whose head was unified with a_i during the given derivation.

Figure 3 shows Skeleton(5) of Example 2. The variable Y of $a(Y)$ in node 1 is annotated as output, since it would be ground at success of $a(Y)$, and $a(Y)$ is a head atom. For the same reason X in node 2, Y in node 3 and in node 4 are annotated as output. The variable Y in node 5 is ground at call, so it is annotated as input.

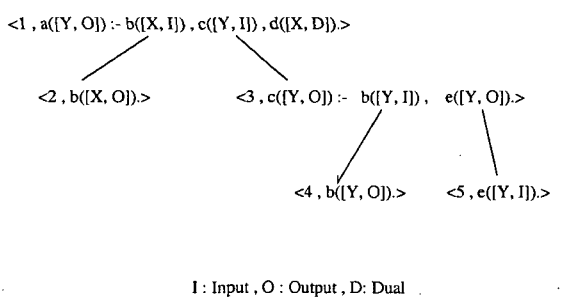


Figure 3: Skeleton(5) for Example 2.

We can refer to the k-th *argument position* in each node of the skeleton by the tuple $(Mark, i, k, V)$, where $Mark$ is the identity number of the node, i is the number of the predicate in the clause (from 0 to m), k is the argument position of a_i and V is the set of variables at this position. If V also contains Input and Output variables it is annotated as Dual.
Denote $Pos(S)$ the set of argument positions of the Skeleton(n) S.

As can be seen from the definition of the Augmented SLD-tree and $Skeleton(n)$, there is an one to one correspondence between the nodes belonging to one branch of the SLD-tree (identified by n) and the nodes of the corresponding $Skeleton(n)$. This correspondence is based on the fact that both structures describe the same derivation for a goal step by step. In our formalism the Mark of a node highlights this correspondence.

Let T be an SLD-tree for the goal g, $n \in nodes(T)$ a leaf of T, and S the $Skeleton(n)$. Then, there is a map from the nodes of S to the nodes of T such that:
$\phi : nodes(S) \rightarrow nodes(T)$
$< Mark, \nu(p : -a_1, \cdots, a_m) > \rightarrow < Mark, (p, R), pred\_def\_ref(p) >$.
If $S' \subseteq nodes(S)$ then denote $\phi(S')$ the corresponding subset of $nodes(T)$ such that $\phi(S') = \{\phi(n) | n \in S'\}$.
For $n \in nodes(T)$ let $\phi^{-1}(n) = m \in Pos(S)$, such that $\phi(m) = n$.

Now we are ready to define the Proof Tree Dependence Graph.

3.3 Proof Tree Dependence Graph

We would like to represent the data flow of a derivation tree. In a logic program data can be transferred in two ways: firstly from one clause to another via unification, and secondly within a clause multiple occurrence of variables result in data dependences [3, 4]. The following definition reflects these conditions.

Definition 4 Proof Tree Dependence Graph (PTDG: $T_{g,n} = (Pos(S), \sim_T)$)
Let T be an SLD-tree for the goal g, $n \in nodes(T)$ a leaf of T and S the Skeleton(n), $\beta, \delta \in Pos(S)$.

- *The nodes of PTDG are the elements of $Pos(S)$.*
- *$\beta \sim_T \delta$ iff one of the following conditions holds:*
  1. *β and δ have common variable in their variable set V (**local edge**)*
  2. *the predicate of δ was unified with the predicate of β, and β and δ are both the k-th argument position of their predicate (**transition edge**).*

It follows directly from the definition that the dependence graph is constructed only for one branch of the SLD-Tree (identified by n), for Skeleton(n). But of course we can construct a PTDG for every Skeleton(n) (n is a leaf node of T), that is for every branch of the SLD-tree T. Figure 4 shows the PTDG for $Skeleton(5)$.

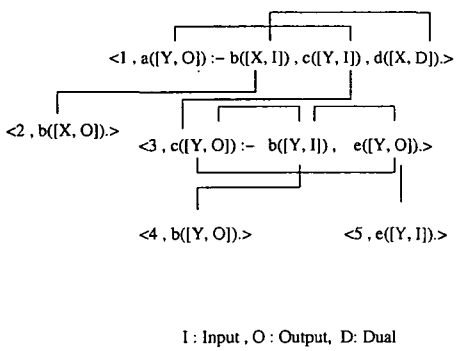


Figure 4: The Proof Tree Dependence Graph for $Skeleton(5)$

3.4 Directed Proof Tree Dependence Graph

As mentioned earlier we would like to better approximate the implicit data flow by introducing directionality using an annotation technique. The annotations can be collected during the execution of the program. Based on this annotation the Proof Tree Dependence Graph can be directed because the data flows from an Input position to an Output one via a local edge, and from an Output position to an Input one via an transition edge. This can be expressed more precisely in the following definition.

Definition 5 Directed Proof Tree Dependence Graph
Let $T_g = (Pos(S), \sim_T)$ be a proof tree dependnece graph, $\alpha, \beta \in Pos(S)$. Then the **directed proof tree dependence graph** *is $\vec{T}_{g,n} = (Pos(S), \rightarrow_T)$, where*

1. *$\alpha \rightarrow_T \beta$ if $\alpha \sim_T \beta$, $\sim_T$ is a local edge and one of the following conditions holds:*
   - *α is an Input position and β is an Output position*
   - *α is a Dual position and β is an Output position*
   - *α is an Input and β is a Dual position*
   - *α is a Dual and β is a Dual position (in this case $\alpha \rightarrow_T \beta$ and $\beta \rightarrow_T \alpha$)*
2. *$\alpha \rightarrow_T \beta$ if $\alpha \sim_T \beta$, the positions being connected by a transition edge and satisfying one of the following conditions:*
   - *α is an Output position and β is an Input position*
   - *α is a Dual and β is a Dual position (in this case $\alpha \rightarrow_T \beta$ and $\beta \rightarrow_T \alpha$)*

It is quite easy to check the validity of these rules. It is possible to define more precise conditions to direct the edges (referring to the textual occurrences of the positions), but it would be too complicated to present them here. Our experience shows that the use of these rules (which permit in some cases non realisable data flow) gives good results. Our slicing algorithm applies to this Directed Proof Tree Dependence Graph. The Directed Proof Tree Dependence Graph for Skeleton(5) is depicted in Figure 5.

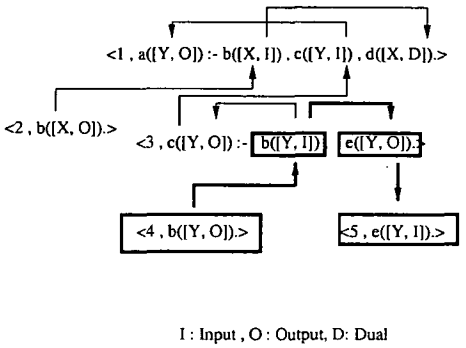


Figure 5: The Directed Proof Tree Dependence Graph and the data flow slice (see Section 3.5) with respect to $(5, 0, 1, \{Y\})$

3.5 General Data Flow Slice

In this section a *general slice definition* is given, which shows that a given argument position of Skeleton(n) which other argument positions depends on (from the aspect of data flow).

Definition 6 Slice$(T_{g,n}, \alpha)$
Let P be a logic program, T a SLD-tree for the goal g, $n \in nodes(T)$ a leaf of T, S Skeleton(n) and $\vec{T}_{g,n} = (Pos(S), \rightarrow_T)$ the corresponding Directed Proof Tree Dependence Graph. Let $\alpha \in Pos(S)$.
A **slice** *$(T_{g,n}, \alpha)$ over $\vec{T}_{g,n}$ with respect to α*

- *is a subgraph of $\vec{T}_{g,n}$*
- *a node $\beta \in Pos(S)$ is in the slice iff $\beta \rightarrow_T^* \alpha$*

This is a general data flow slice definition that is valid for one derivation path of the SLD-tree which may be a failed branch.

Figure 5 shows $slice(T_{a(y),5}, (5, 0, 1, \{Y\}))$. The argument of $e(Y)$ in node 5 $((5, 0, 1, \{Y\}))$ is an Input position, the slice being constructed with respect to this position. The set of all positions from which there is a directed path to this argument position (this is the slice with respect to $(5, 0, 1, \{Y\})$) contains the argument position of $e(Y)$ and $b(Y)$ in node 3, and $b(Y)$ in node 4.

The data flow slice given in [9] is a special case of this slice definition, which is defined only for the success branch of the SLD-tree.

4 Debug Slice of Logic Programs

As mentioned before our aim is to combine our slicing technique with an algorithmic debugging tool [14] in order to locate the source of a bug with fewer user interactions by avoiding posing unnecessary questions. In [9] a data flow slice was defined for the success branch of the SLD-tree (Trace-tree). We extended the data flow slice definition to the full SLD-tree (Trace-tree), based on the Directed Proof Tree Dependence Graph.

To take care of the control flow mentioned in Subsection 4.1 we specify the *Potentially Dependent Predicates Set (PDPS)* which contains the predicates that actually did not affect the selected argument, but could have done so had they been evaluated differently (i.e. had they succeeded or failed).

Lastly in Subsection 4.2 the *Debug slice* is defined on the Augmented SLD-tree (Trace tree) which includes the Potentially Dependent Predicates, their associated data dependences and the predicates affected by some cut.

4.1 Potentially Dependent Predicates

Sometimes we cannot find the source of a bug just by analyzing the data flow for the success branch of the Trace-tree. So we have to examine which predicates might cause a branch of the Trace-tree to fail, or what would have happened if a predicate had succeeded but actually failed, or if it should have failed but actually succeeded. We concentrated on the leftmost (goal) predicate of an SLD node so the slice is defined for these predicates. The following definition covers these cases.

Definition 7 Potentially Dependent Predicate (PDPS)
Let P be a logic program, T the Trace-tree for the goal g. A leftmost (selected) predicate in a node of T is in the **Potentially Dependent Predicate Set (PDPS)** *if it actually did not affect the value of an argument of a predicate in the success branch of T, but could have affected it had its boolean outcome been different.*

In the following we try to identify those predicates which satisfy this condition.

Lemma 1 *Let P be a logic program, T the Trace-tree for the goal g. Then PDPS= {The predicates of the success branch of T} ∪ {The predicates of the failed leaves of T}.*

Proof To prove the validity of this Lemma we have to demonstrate that these predicates really satisfy the condition of Definition 7 while the other predicates of T do not. To achieve this, we classify the predicates of T in such a way that the categories cover all predicates belonging to T. Notice that we use "the selected variable" expression but it could have been any variables of the program whose values do not satisify our expectations (i.e. where a bug was manifested). The

PDPS is the same for every selected variable, so it is created for a Trace tree built up for a given goal.

- **If a predicate should have failed but actually succeeded**

  1. If this predicate (selected goal) belongs to the success branch of the Trace-tree, then its boolean outcome could have affected the value of the selected variable (argument), so it could have been the source of the bug. We notice that this situation caused the modification of the structure of the Trace-tree.
  2. If this predicate belongs to a failure branch of the Trace-tree, then its boolean outcome could not have affected the value of the selected variable because if had it failed it would then have caused the pruning of the subtree below this predicate. But this would have not modified the structure of the other parts of the Trace-tree.

- **If a predicate should have succeeded but it failed**

  Then this predicate is a leaf of a failed branch of the Trace-tree (because these are the only failed predicates). Its boolean outcome could have affected the value of the selected variable because it might have modified the structure of the Trace-tree.

To extend this lemma we notice that if the user had found a bug in a success branch of the SLD-tree which was not the first one, then the predicates of the previous success branches did not belong to the PDPS because if they had failed it would not have affected the structure of the later branches of the SLD-tree.

Example 4 *The PDPS of the Trace-tree in Figure 1 is the following (The nodes are identified with their marks):*
$PDPS$=$\{1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14\}$.

4.2 Debug Slice

In this section our main result *the Debug slice* is specified based on the definitions and Lemma of the previous sections. The Potentially Dependent Predicate Set of a Trace-tree (for a logic program P and goal g) includes all those predicates whose boolean outcome may affect the value of an argument in a success branch's predicate. The Debug slice deals with control dependences as well. The Debug slice is a set of predicates which contains the Potentially Dependent Predicates [Section 4.1], their data dependences [Section 3.5] and the predicates affected by some *cut*. Since the Debug slice contains all predicates of the success branch of T, the Debug slice is the same with respect to every selected argument. Hence the Debug slice is defined for a logic program P and goal g.
An interesting question is the effect of cuts. If there is a node in the Trace-tree whose leftmost goal is cut($Mark$) we remove all descendants of this node to the

right of the path W up to the node marked $Mark$. We denote this kind of path by cut(W).

An informal definition for the Debug slice is the following.
Let P be a logic program, T be the Trace-tree for the goal g.
The *Debug slice of* T consists of the following predicates:

1. The predicates of the Potentially Dependent Predicate Set (PDPS)
2. The predicates specified by the data flow of the predicates of PDPS
3. The predicates that belong to some cut(W) of T

1. The predicates of the Potentially Dependent Predicate Set (PDPS)

The Potentially Dependent Predicate Set of an Trace-tree includes all predicates whose boolean outcome may affect the value of an argument in a success branch's predicate. So these predicates affect the control dependences. Lemma 1 describes those predicates which belong to the PDPS.
We would like to extend this set. But then we must first see how is it possible to describe the new predicates that are introduced by the data flow and then see why $cut(W)$ belongs to the Debug slice too.

2. The predicates specified by the data flow of the predicates of PDPS

Since PDPS consists of two subsets, the first point is dealt with by examining two cases in turn.

1. **The predicates that belong to the success branch of T**

   Here the data flow does not introduce new predicates into the Debug slice as the data flow slice is valid for one given branch of the Trace-tree (T), and all predicates of the success branch of T are in the Debug slice.

2. **The predicates of the failed leaves of T**

   Let $n \in PDPS$ such that n is a leaf node of a failed branch of T, S the Skeleton(n), $\vec{T}_{g,n}$ the corresponding directed Proof Tree Dependence Graph (see Section 3.4), and $\phi^{-1}(n)$ the corresponding node of S (see Section 3.2). Suppose that $\phi^{-1}(n)$ is labeled by $< n, p : -a_1, \cdots, a_m >$.
   Next, construct $slice(\vec{T}_{g,n}, \alpha)$ for every $\alpha \in Pos(S)$ such that α is an argument position belonging to the head predicate p.
   Let $H = \cup_{n,\alpha}\{k \in nodes(S) \mid k$ has at least one head argument position in $slice(T_{g,n}, \alpha)$, α is an argument position of p, n is a failed leaf of $T\}$.
   Afterwards, map H back to the Trace tree ($\phi(H)$). So $\phi(H)$ contains those predicates which are specified by the data flow of the failed leaves of T.
   Let the set of these predicates be denoted by S_1.
   For example, one can see in Figure 5 that $n = 5$ is a failed leaf of the Trace tree (Figure 1), then $\phi^{-1}(5)$ is $< 5, (e(Y, I) >$. As the clause contained in this node is a fact, the head predicate is $e(Y)$, which has one argument position

Y. Constructing the slice with respect to this argument position it contains head argument position from node 4 and 5. So $\phi(S1)$ in this case contains nodes 4 and 5 of the Trace tree in Figure 1.

Now we will address the second point.

3. The predicates that belong to some cut(W) of T

If there is a node in the Trace-tree whose **leftmost goal is cut(Mark)** we remove all descendants of this node to the right of the path W up to the node marked $Mark$. So a cut may effect the control dependences and those nodes that belong to W have to be added to the Debug slice. The removed part of the Trace-tree might have the right solution.

Let the set of these predicates be denoted by S_2.

Example 5 *In Example 3 if we had not had cut in clause 3, we would have got* $X = 1$, *so* $< 5, e(X) >$ *and* $< 6, h(X) >$ *would have affected the value of the variable* X *in* $a(X)$, *but this would not have shown up in data flow analysis.*

Definition 8 *The* **Debug slice** *of an Augmented Proof Tree for a goal* g *is the following set:*
Debug slice= $PDPS \cup S_1 \cup S_2$.

Example 6 *In Example 1 the PDPS contains the buggy predicate* $A > 0$, *so* $A > 0$) *is in the Debug slice, but the data flow slice does not have it because this predicate belongs to a failed branch of the SLD-tree for the goal* $p(0, X)$.

Example 7 *We will now go on with Example 2.*

In order to construct the Debug slice we furnish the sets PDPS, S_1, S_2.

1. We know (see Section 4.1) that **PDPS** $= \{1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14\}$ for the Augmented SLD-tree in Figure 1.
2. To get S_1 we have to construct a Skeleton(n) for every $n \in \{1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14\}$ and the corresponding Proof Tree Dependence Graphs. We also have to specify a $slice(\vec{T}_{g,n}, \alpha)$ for every α argument position of n in the Proof Tree Dependence Graph and to state every node of T that belongs to these slices. Figure 5 shows the Proof Tree Dependence Graph for Skeleton(5) and $slice(\vec{T}_{g,n}, (5, 0, 1, \{Y\}))$. We urge the reader to construct all the slices for each argument position of $\{1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14\}$. In our case the only node in T specified by these slices is node 4. So $\vec{S}_1 = \{4\}$.
3. We had no cut in this example, so $\vec{S}_2$ is empty.

Then, **Debug slice**= $\{\{1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14\} \cup \{4\} \cup \{\}\} =$
$\{1, 2, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14\}$

In this example the only nodes that are not in the Debug slice are 3 and 7. The Debug slice of the Trace-tree built for Example 2 is emphasised and framed on Figure 1.

5 Prototype Implementation

We have developed a prototype in Sicstus Prolog using the complete framework described for slicing Prolog programs. The implementation handles specific programming techniques (cut, if-then, OR). The Trace-tree is constructed from the call ports of the trace given by the Interpreter. The slice with respect to a node of the Trace-tree is created. The slicing technique, if desired, can be combined with Shapiro's debugging method [14, 19]. To first approximation the slice is built up for the success branch of the Trace-tree. The slice is then constructed with respect to the given argument position, and during a debugging session the system asks for the validity of just those nodes that are in the slice. If it is unsuccessful in locating the source of the bug, the Debug slice is constructed and the user can then request data flow information as well for any leaf node predicate of the Debug slice. A graphical interface draws the Trace-tree and highlights those nodes that are included in the data flow slice and in the Debug slice [10].

We tested our Debug Slice algorithm on several small Prolog programs. These programs can produce big fail branches for some inputs. The test results are shown in the Table below. We examined the number of nodes and arguments in the whole Trace tree, in the success and failed branch of the Trace tree, and lastly in the Debug slice.

	Complete Tree		Success Branch		Failed Branch		Debug Slice	
	Nodes	Arg.	Nodes	Arg.	Nodes	Arg.	Nodes	Arg.
1.	14	14	6	6	8	8	13	13
2.	15	26	6	9	9	17	7	10
3.	19	39	7	11	12	28	11	20
4.	34	64	15	15	19	49	23	39
5.	267	618	11	13	256	605	16	23
6.	316	769	15	15	301	754	23	39
7.	520	1393	24	49	496	1344	181	423
8.	622	1240	6	9	616	1231	7	10
9.	1142	2687	5	9	1137	2678	7	13

The test results demonstrate that if the number of the failed branch's nodes is high and the data flow slices for the failed branch's leaf predicates do not contain too many predicates, then the Debug slice is significantly smaller than the whole Trace tree. The test results of course depend on the size and type of input, as well. The Debug slice method handles types of bugs which the conventional data-flow slice technique misses. Certain types of bugs were found during testing which were missed by the data-flow slice but were identified using the Debug slice method, as they appeared in the failure branches of the SLD-tree. These types include cases, when:

- a cut is mis-placed.
- a failed predicate is mis-printed (its name or arity) or a condition $(<, >, =)$ is failed.

- a wrong data value has reached the failed node; so in the data-flow from the root to the failed node, a wrong constant value, a mis-printed predicate or a failed condition has appeared.

We notice that the system finds only those mis-printed predicates of the failure branches of the SLD-tree which occur in the data-flow of a failed predicate, or are affected by a cut.

6 Related Work and Discussion

While program slicing has been widely studied for imperative programs [23], relatively few papers have dealt with the problem of slicing logic programs [9, 20, 22, 27].

Gyimóthy and Paaki present in [9] a specific slicing algorithm for sequential logic programs in order to reduce the number of user queries of an algorithmic debugger. But they only analyzed the data dependences for the success branch of the SLD-Tree (Trace-tree). Sometimes it is insufficient to locate the source of a wrong solution because the cause of the erroneous result may also be an invalid the proof tree structure. We solved this problem by dealing with control dependences, as well. So the data flow slice given in [9] is a special case of our approach.

Schoening and Ducasé have proposed a backward slicing algorithm for Prolog which produces executable slices [20]. An executable slice is usually less precise than a general slice [23], and their algorithm is only applicable to a limited subset of Prolog programs. Our aim was to develop a tool for debugging Prolog programs that also handles specific programming techniques.

In [27] Zhao at al. presented a new program representation called the argument dependence net for concurrent logic programs in order to produce static slices at the argument level. Dynamic slicing usually produces more precise slices than static ones because it only considers a particular execution of a program. We chose the dynamic version because our application focuses on debugging .

In [17] Pereira and Calejo examined the wrong solution suspect set (WSS) and the missing solution suspect set (MSS). It is possible to refine WSS using our general slice definition.

In [22] a dynamic slicing method was presented for constraint logic programs based on variable sharing and groundness analysis. In the paper the declarative formulation of the slicing problem for constraint logic programs was also described.

The Debug slicing method for Prolog programs was introduced in this paper. This slicing technique is very appropiate for debugging because it deals with control dependences as well. This slicing method will be integrated into the IDTS interactive algorithmic debugging tool [14]. This tool employs an improved version of Shapiro's debugging method [19] for identifying a buggy clause. By using slicer modules the number of user interactions can be reduced during the debugging process. The data flow slice is usually much smaller than the Debug slice, so in the first step we can try to locate the bug in the data flow slice. However it may happen that the data flow slice does not contain the buggy clause. In this case the debugging process has to be extended to the nodes of Debug slice. We tested our slicing tool

on small Prolog programs [10]. Now we plan to improve the implementation of the tool so that it will be able to analyze real-sized Prolog programs. We also would like to compare the size of different slices of big SLD-trees.

References

[1] D.C.Atkinson and W.G.Griswold: The Design of Whole-program Analysis Tools. *In Proceedings of the 18th International Conference on Software Engineering*, pages 16-27, Berlin, March 1996.

[2] S. Bates, S. Horwitz: Incremental program testing using program dependence graphs. *In Proceedings of Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages* , pages 384-396, Charleston, 1993.

[3] J. Boye, J. Paakki, J. Maluszyński: Dependency-Based Groundness Analysis of Functional Logic Programs. *Research Report LiTH-IDA-R93-20*, Department of Computer and Information Science, Linköping University, 1993.

[4] J. Boye, J. Paakki, J. Maluszyński: Synthesis of Directionality Information for Functional Logic Programs. *In Proceedings of 3rd International Workshop on Static Analysis*, LNCS 724, Springer-Verlag, pages 165-177, Padova, 1993.

[5] J. Cheng: Dependence Analysis of Parallel and Distributed Programs and its Applications. *In Proceedings of International Conference on Advances in Parallel and Distributed Computing (IEEE-CS)*, 1997.

[6] P. Deransart and J. Małuszyński: *A grammatical view of logic programming.* The MIT Press 1993.

[7] J. Ferrante, K.J. Ottenstein and J. D. Warren: The Program Dependence Graph and its Uses in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3): pages 319-349, July 1987.

[8] T. Gyimóthy, Á. Beszédes and I. Forgács: An Efficient Relevant Slicing Method for Debugging. *In Proceedings of 7th European Software Engineering Conference (ESEC'99)*, pages 303-322, Toulouse, France, September 1999.

[9] T. Gyimóthy and J. Paakki: Static Slicing of Logic Programs.*In Proceedings of Second International Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, pages 85-105, Saint Malo, France, May 1995.

[10] L. Harmath, Gy. Szilágyi, T. Gyimóthy and J.Csirik: Debug Slicing of Logic Programs. *Technical Reports 99/04*, RGAI, Szeged, 1999, *www.inf.u-szeged.hu/rgai/.*

[11] S. Horwitz, J. T. Reps and D. Binkeley: Interprocedural Slicing Using Dependence Graphs. *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 23(7), pages 35-46, July 1988.

[12] S. Horwitz and T. Reps: The Use of Program Dependence Graphs in Software Engineering. *In Proceedings of 14th International Conference on Software Engineering*, pages 392-411, Melbourne Australia, May 1992.

[13] Kamkar M.: Interprocedural Dynamic Slicing with Applications to Debugging and Testing. Linköping Studies in Science and Technology - Dissertation No. 297, Department of Computer and Information Science, Linköping University, 1993.

[14] G. Kókai, L. Harmath, T. Gyimóthy: Algorithmic Debugging and Testing of Prolog Programs. *In Proceedings of the 14th International Conference on Logic Programming (ICLP'97). Eighth Workshop on Logic Programming Environments*, pages 14-21, Leuven Belgium, July 1997.

[15] B. Korel and J. Rilling: Application of Dynamic Slicing in Program Debugging. *In Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG '97)*, Linköping, Sweden, May 1997.

[16] K. J. Ottenstein and L. M. Ottenstein: The Program Dependence Graph in a Software Development Environment. *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, 1984*, SIGPLAN Notices, 19(5), pages 177-184, May 1984.

[17] L.M. Pereira and M. Calejo : A Framework for Prolog Debugging, *ICLP/SLP*, pages 481-495, 1988.

[18] U. Nilsson and J. Maluszyński: Logic, Programming and Prolog. *John Wiley and Sons*, 1990.

[19] E.Shapiro: Algorithmic Debugging. *The MIT Press*, 1993.

[20] S. Schoenig and M. Ducassé: A Backward Slicing Algorithm for Prolog. *In Proceedings of 3rd International Static Analysis Symposium (SAS'96)*, LNCS 1145, Springer-Verlag, pages 317-331, 1996.

[21] C. Steindl: Intermodular Slicing of Object-oriented Programs. *In International Conference on Compiler Construction (CC'98)*, 1998.

[22] Gy. Szilágyi, T. Gyimóthy, J. Maluszyński: Slicing of Constraint Logic Programs. *Linköping University Electronic Press* 1998/020, *www.ep.liu.se/ea/cis/1998/002.*

[23] F. Tip: A survey of Program Slicing Techniques. *Journal of Programming Languages*, Vol.3., No.3, pages 121-189, September, 1995.

[24] U. Nilsson and J. Maluszyński: Logic, Programming and Prolog. *John Wiley and Sons*, 1990.

[25] M. Weiser: Programmers use slices when debugging. *Communications of the ACM*, pages 446-452, July 1982.

[26] M. Weiser: Program Sicing. *In Proceedings of Transactions on Software Engineering (IEEE)*, pages 352-357, July 1984.

[27] J. Zhao, J. Cheng and K. Ushijima: Slicing Concurrent Logic Programs . *In Proceedings of Second Fuji International Workshop on Functional and Logic Programming*, pages 143-162, 1997.

Acta Cybernetica15(2001) 279–304.

Using Decision Trees to Infer Semantic Functions of Attribute Grammars *

Szilvia Zvada† and Tibor Gyimóthy†

Abstract

In this paper we present a learning method called LAG (Learning of Attribute Grammar) which infers semantic functions for simple classes of attribute grammars by means of examples and background knowledge. This method is an improvement on the AGLEARN approach as it generates the training examples on its own via the effective use of background knowledge. The background knowledge is given in the form of attribute grammars. In addition, the LAG method employs the decision tree learner C4.5 during the learning process. Treating the specification of an attribute grammar as a learning task gives rise to the application of attribute grammars to new sorts of problems such as the Part-of-Speech (PoS) tagging of Hungarian sentences.

Here we inferred context rules for selecting the correct annotations for ambiguous words with the help of a background attribute grammar. This attribute grammar detects structural correspondences of the sentences. The rules induced this way were found to be more precise than those rules learned without this information.

1 Introduction

Attribute grammars were introduced in [11] as a formalism for the specification of the semantics of program languages (see [1, 4]). They can be considered as an extension of context-free grammars in the sense that attributes and their semantic functions are related to the symbols of the grammar. An attribute is a named property with given values and a semantic function computes its value based on the values of other attributes. A semantic functions may be complex, therefore the specification of an attribute grammar may be a laborious task. Hence a tool which is able to complete a partially given attribute grammar by means of examples would be very useful. The term "partially given" here means that some of the attributes might lack semantic function. The task is to define these unknown semantic functions.

Based on the correspondence of the nonterminals of attribute grammars and the predicates of logic programs (see [5, 6, 17]), we can apply many of techniques to

*This work was supported by the grants OTKA T52721 and IKTA 8/99.

†Research Group on Artificial Intelligence, University of Szeged, H-6720 Szeged, Aradi vértanúk tere 1, Hungary. e-mail: {zvada,gyimi}@sol.inf.u-szeged.hu

attribute grammars, which techniques were originally developed for logic programs. For instance, viewing the specification of an attribute grammar as a learning task, learning methods presented in the framework of inductive logic programming (ILP, see [12, 14]) can be used to solve this task.

ILP is an active research area of machine learning that studies the definitions of logic programs from examples and the presence of background knowledge. Since examples and background knowledge are expressed in first-order logic, ILP methods can be employed to learn relational and recursive definitions. This last property makes ILP methods very promising for the attribute grammars, because recursive rules often emerge in attribute grammars as well.

This fact suggested the use of a similar learning approach in the AGLEARN method ([8]) to that one employed in the ILP learning system called LINUS ([12]), but in a different representational framework. That is, the learning task and background knowledge is represented in the form of attribute grammars instead of logic programs. The task of AGLEARN is to complete the specifications of an S-attributed or an L-attributed grammar based on positive and negative examples. These examples contain strings derived from the target nonterminal, the attributes of this target nonterminal being evaluated in these strings. The main idea behind AGLEARN is converting the learning task into a propositional form then inferring the unknown semantic function with the help of a propositional learner.

In this paper we introduce the LAG approach which is based on the AGLEARN method, but it uses the given background knowledge more effectively and employs the C4.5 decision tree learner (see [19]) instead of a propositional learner. Doing this allows treating the learning task as a classification problem with multiple classes. The robustness of the C4.5 for classification problems has already been demonstrated. Another important difference between the AGLEARN and LAG methods can be seen in the handling of the training examples. In the case of the former, the user has to explicitly define each training example in advance. With the latter, the input of the LAG system consists of strings taken from the language generated by the partially given attribute grammar. The LAG system builds the decorated decision trees of these strings and evaluates the attribute instances during the tree traversals. Whenever an attribute instance with no semantic function is computed its value is defined by the user ("oracle"). Hence even a few strings can produce a large number of training examples.

The LAG method is applied to the Part-of-Speech tagging of Hungarian sentences. The task here is to distinguish the different morphologic classes of a word, as in the case of "múlt"[1], which might be annotated by a verbal, noun or adjectival tag. The tagging of Hungarian texts is very difficult due to the rich morphology of the language. Our method has been applied in order to infer the rules for selecting the contextually correct tags. The input data set, a corpus with about 100 000 pre-tagged words ([7, 16]), is employed for training and testing. The background attribute grammar determines some structural information of the parts of sentences

[1] *múlt (verb) -passed*
múlt (noun) - past
múlt (adjective) - past, last

like subject phrase and predicate phrase. Based on the latter the training data sets for the C4.5 system are generated. By using the training sets decision rules are inferred for ambiguous words. The experimental results show (see also [2, 9]) that the use of even simple attribute grammars as background knowledge yields more effective rules than a method which lacks this structural information.
In Section 2 the key definitions relating to the learning of attribute grammars are introduced, while in Section 3 the LAG method itself is discussed. Afterwards, in Section 4 a brief overview of the PoS tagging problem and the application of the LAG method is presented. The accuracy of the C4.5 and LAG approaches is compared in Section 5. In the final section, the conclusions are drawn and suggestions for future research are offered.

2 Preliminaries

In this section we introduce the terminology and notations used in this paper.

2.1 Attribute grammars

Attribute grammars were introduced in [11] as an extension of **context-free grammars** (**cfg** onwards) for specifying static semantics of programming languages, such as type-checking and name-analysis during syntax-directed parsing. This is achieved by attaching attributes (named properties with given values) to the symbols of the grammar. During the parsing a derivation tree based on the underlying *cfg* is constructed. In this tree nodes and leaves are labeled by nonterminals and terminals of the *cfg*, respectively. The **instances** of the attributes appear in this tree along with the grammar symbols which they are related to. This tree is called **decorated derivation tree** or simply **ddt**.
The value of an attribute instance is defined by its semantic function during the traversal of the ddt. The value of an attribute is determinable iff the values of all the attributes in the argument of the semantic function have already been computed. In this way the semantic functions define dependency relations among the attributes. The attributes transmit information within the ddt in two directions: from the root to the leaves, where they are named **inherited attributes**, or backwards, where they are called **synthesized attributes**.
Before we formally define the learning task for attribute grammars, let us first consider the definitions and notations of attribute grammars (cf. [1]).
An **attribute grammar** (briefly **ag**) is a four tuple $AG = (G, SD, AD, R)$ which consists of the following components:

- an underlying *cfg* $G = (V_N, V_T, P, S)$
- a semantic domain $SD = (\mathcal{T}, \mathcal{F})$ consisting of a set $\mathcal{T}$ of the domains of attributes and a set $\mathcal{F}$ of functions over the attributes: $type_1 \times \cdots \times type_m \rightarrow type_0$ for $type_i \in \mathcal{T}$ $(0 \leq i \leq m)$.
  (If $type_0 = \{true, false\}$ then we talk about *relations*.)

- an attribute description is a triple $AD = (Inh, Syn, \tau)$ where Inh and Syn are finite, disjoint sets of **inherited** and **synthesized attributes**, respectively. $Attr = Inh \cup Syn$ is the set of all attributes of AG. Let $X.a$ denote an attribute $a \in Attr$ attached to the grammar symbol $X \in V_N \cup V_T$. The set $Inh(X)$ and set $Syn(X)$ consist of the inherited and synthesized attributes of the symbol X, respectively. τ is a function mapping attributes to their types (domains) such that $\tau : Attr \to \mathcal{T}$.
- a set $R = \{R(p) \,|\, p \in P\}$ consisting of finite sets $R(p)$ of **semantic functions** which are associated with the production $p : X_0 \to X_1 \ldots X_{m_p}$. An **occurrence** of an attribute $X_k.a$ in the production p is denoted by $X_k^p.a$. The set

  $$DO(p) = \{X_0^p.s \in Syn(X_0)\} \cup \{X_k^p.i \in Inh(X_k) \text{ with } 1 \le k \le m_p\} \text{ and}$$

  $$UO(p) = \{X_0^p.i \in Inh(X_0)\} \cup \{X_k^p.s \in Syn(X_k) \text{ with } 1 \le k \le m_p\}$$

  of **defined attribute occurrences** and **used attribute occurrences** of p, respectively, are assigned to every production $p \in P$. For every $X_k^p.a \in DO(p)$ there is exactly one semantic function given in $R(p)$

  $$X_k^p.a = f\left(X_{k_1}^p.a_1, \ldots X_{k_s}^p.a_s\right)$$

  with $(f : \tau(a_1) \times \cdots \times \tau(a_s) \to \tau(a)) \in \mathcal{F}$ and $X_{k_i}^p.a_i \in UO(p)$ for $1 \le i \le s$. Then we say that $X_k^p.a$ depends on $X_{k_i}^p.a_i$, for $1 \le i \le s$. (Note that if $s = 0$ the function is a constant $c \in \tau(a)$.)

In several applications it is useful to attach a special, synthesized, boolean attribute *accept* to the start symbol S of the underlying *cfg*. Using the attribute *accept* we can define the language generated by an attribute grammar like so:

$$Lang(AG) = \{w \,|\, w \in Lang(G) \text{ and } S.accept = true \text{ in the ddt of } w\}.$$

Let $AG = (G, SD, AD, R)$ be an *ag* with an underlying *cfg* $G = (V_N, V_T, P, S)$, a semantic domain $SD = (\mathcal{T}, \mathcal{F})$ and an attribute description $AD = (Inh, Syn, \tau)$. Furthermore, let t be a *ddt* and n_0 be a node of t, which is labelled by $X_0 \in V_N \cup V_T$. The set $Inh(n_0) = \{n_0.i | X_0.i \in Inh(X_0)\}$ of **inherited attribute instances** and the set $Syn(n_0) = \{n_0.s | X_0.s \in Syn(X_0)\}$ of **synthesized attribute instances** are associated with the node n_0. Thus $Inst(n_0) = Inh(n_0) \cup Syn(n_0)$. (Note that $\tau(n_0.i) = \tau(X_0.i) = \tau(i)$ holds for any $n_0.i \in Inst(n_0)$.)
Further, let the production $p : X_0 \to X_1 \ldots X_{m_p}$ be applied at node n_0. Then $X_1, \ldots, X_{m_p}$ label the successors $n_1, \ldots, n_{m_p}$ of n_0, from left to right, respectively. Let

$$DI(n_0, p) = \{n_k.a \,|\, X_k^p.a \in DO(p) \text{ with } 0 \le k \le m_p\} \text{ and}$$

$$UI(n_0, p) = \{n_k.a \,|\, X_k^p.a \in UO(p) \text{ with } 0 \le k \le m_p\}$$

be the set of **defined attribute instances** and **used attribute instances** of n_0, respectively. Then an instance $n_l.a$ of the defined attribute occurrence $X_l^p.a$ is determined by $f(n_{k_1}.a_1, \ldots, n_{k_m}.a_m)$, where $n_{k_1}a_1 \ldots n_{k_m}a_m \in UI(n_0, p)$ are the

instances of the attribute occurrences $X^p_{k_1}.a_1, \ldots, X^p_{k_m}.a_m \in UO(p)$ and f is the interpretation of the semantic function

$$X^p_l.a = f\left(X^p_{k_1}.a_1, \ldots X^p_{k_m}.a_m\right).$$

An attribute instance $n_{k_0}.a$ depends on the attribute instances $n_{k_i}.a_i$, for $1 \leq i \leq m$. It is clear that an attribute instance $n_l.a$ can be computed if all attribute instances on which it depends have already been evaluated.
An *ag* is *circular* if it has a ddt such that there is a circular dependency among the attribute instances. Otherwise an *ag* is *non-circular*. Here we consider two subsets of the non-circular *ags*, namely the L-attributed and S-attributed grammars.
An *ag* is **L-attributed** if all attribute instances of an arbitrary ddt of this *ag* can be evaluated in one left-to-right tree traversal. The left-to-right traversal and the attribute evaluation are described by the following procedure:

```
proc tree_traversal(node : n0);
    begin
        for i := 1 to mp do
            begin
                eval(Inh(ni));
                tree_traversal(ni);
            end;
        eval(Syn(n0));
    end;
```

One can formulate conditions for the L-attributed property. Let $X^p_l.a$ be a defined attribute occurrence of the production p and $X^p_l.a = f\left(X^p_{k_1}.a_1, \ldots X^p_{k_s}.a_s\right)$. Then the *ag* is L-attributed if the following conditions hold for each defined attribute occurrence (see Figure 2.1):

- if $X^p_l.a$ is an inherited attribute occurrence then $X^p_{k_i}.a_i \in Inh(X^p_0)$ or $X^p_{k_i}.a_i \in Syn(X^p_{k_i})$, with $1 \leq i \leq s$ and $1 \leq k_i < l$. This means that an inherited attribute occurrence $X^p_l.a$ may depend on the synthesized attribute occurrences of the rhs symbols $X^p_{k_i}$, that have been defined before than X^p_l. It may also depend on the inherited attribute occurrences of the lhs symbol X^p_0 as shown in Figure 1. Here an inherited attribute occurrence is visualized by a white circle above the respective symbol, whereas a synthesized attribute occurrence is depicted as a black dot below it.

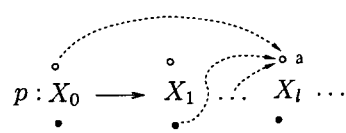


Figure 1: L-attributed dependencies of $X^p_l.a$

- if $X^p_0.a$ is a synthesized attribute occurrence then $X^p_{k_i}.a_i \in Inh(X^p_0)$ or $X^p_{k_i}.a_i \in Syn(X^p_{k_i})$, with $1 \leq i \leq s$ and $1 \leq k_i \leq m_p$. Namely, this

means that an lhs synthesized attribute occurrence $X_0^p.a$ of a rule may depend on synthesized attribute occurrences of rhs symbols and inherited attribute occurrences of the lhs symbol, itself. Figure 2 presents these relations.

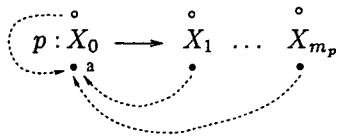


Figure 2: L-attributed dependencies of $X_0^p.a$

Let the set $UO_{L-attr}(X_l^p.a)$ denote the used attribute occurrences of p which fulfill these two conditions with respect to the attribute occurrence $X_l^p.a$.
The other subset of non-circular *ag*s investigated in this paper is the S-attributed grammar. An *ag* is **S-attributed** if solely synthesized attributes are related to the symbols of the grammar. It is clear that the set of S-attributed grammars is a subset of L-attributed grammars.
To help to make these definitions clearer, let us illustrate their use with a concrete example.

Example 1 *The S-attributed* ag $AG_{typ} = (G_{typ}, SD_{typ}, AD_{typ}, R_{typ})$ *defined below determines whether the type of an arithmetical expression is real or integer.*

nonterminals and terminals $V_N = \{Expr, Term, Factor, AddOp, MulOp, \}$
$V_T = \{Integer, Real, =, -, *, /, \lambda\}$

the semantic domain SD_{typ} $\mathcal{T} = \{type_{mode}, type_{op}\}$, *where*
$type_{mode} = \{int, real\}$, *and*
$type_{op} = \{add, sub, mul, div\}$
$\mathcal{F} = \{f_1 : type_{mode} \times type_{mode} \rightarrow type_{mode},$
$f_2 : type_{mode} \times type_{op} \times type_{mode} \rightarrow type_{mode}\}$
where $f_1(x, y) =$ ***if*** $(x = int) \wedge (y = int)$
then *int*
else *real*
$f_2(x, y, z) =$ ***if*** $(x = int) \wedge (y = mul) \wedge (z = int)$
then *int*
else *real*

the attribute descriptions AD_{typ} $Inh = \emptyset$
$Syn = \{mode, op\}$
$Syn(Expr) = Syn(Term) = Syn(Factor) = \{mode\}$
$Syn(AddOp) = Syn(MulOp) = \{op\}$
$\tau(mode) = \{int, real\}$
$\tau(op) = \{add, sub, mul, div\}$

the underlying cfg G_{typ} *and the set* R_{typ} *of semantic functions:*

1, $Expr_0 \rightarrow Expr_1\ AddOp\ Term$
$R(1) = \{\ Expr_0.mode := f_1(Expr_1.mode, Term.mode)\}$

$2,\ Expr \rightarrow Term$
$R(2) = \{ Expr.mode := Term.mode\}$
$3,\ Term_0 \rightarrow Term_1\ MulOp\ Factor$
$R(3) = \{Term_0.mode := f_2(Term_1.mode, MulOp.op, Factor.mode)\}$
$4,\ Term \rightarrow Factor$
$R(4) = \{Term.mode := Factor.mode\}$
$5,\ Factor \rightarrow Integer$
$R(5) = \{ Factor.mode := int\}$
$6,\ Factor \rightarrow Real$
$R(6) = \{ Factor.mode := real\}$
$7,\ Factor \rightarrow (\ Expr\ )$
$R(7) = \{ Factor.mode := Expr.mode\}$
$8,\ AddOp \rightarrow +$
$R(8) = \{ AddOp.op := add\}$
$9,\ AddOp \rightarrow -$
$R(9) = \{ AddOp.op := sub\}$
$10,\ MulOp \rightarrow \times$
$R(10) = \{ MulOp.op := mul\}$
$11,\ MulOp \rightarrow /$
$R(11) = \{ MulOp.op := div\}$

some of the defined and used attribute occurrences:

$DO(1) = \{Expr_0.mode\}$
$DO(2) = \{Expr.mode\}$
$DO(3) = \{Term_0.mode\}$

$UO(1) = \{Expr_1.mode, AddOp.op, Term.mode\}$
$UO(2) = \{Term.mode\}$
$UO(3) = \{Term_1.mode, MulOp.op, Factor.mode\}$

It is immediately apparent that for S-attributed grammars, all the used attribute occurrences satisfy the L-attributed property.
Nevertheless, the specification of semantic functions is not trivial even in the case of L-attributed and S-attributed grammars. The current paper introduces a method which learns the semantic functions of *ags* like these.

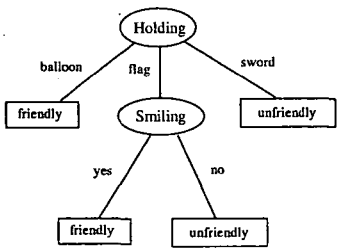
2.2 Inductive learning

The idea of using inductive learning methods to define semantic functions of an attribute grammar was motivated by the parallelism found between the nonterminals of attribute grammars and the predicates of logic programs (see [5, 6, 17]).
In general, an inductive learning method studies a set of positive and negative *training examples* and *background knowledge* in order to infer a hypothesis which approximates the *target concept.* The inferred hypothesis explains the training examples together with the background knowledge such that all positive examples can be 'proved' by it and no negative example can be 'derived' from it. Many inductive learning approaches use an attribute-value language to represent the

examples, background knowledge and the concept to be induced. The most popular of these attribute-value learners are *decision tree learners* used widely in solving classification problems ([13, 19]).
These methods construct decision trees for modelling the target hypotheses from the training examples expressed as attribute-value vectors. In a decision tree, every interior node is labelled with a test over an attribute which is expected to most efficiently classify the current subset of training examples. The possible outcomes of these attribute tests assign a name to the branches descending from the nodes. The leaves show a "class" to which the examples of the current training set belong. The decision trees can be also represented by a set of *decision rules* (see Example 2). The LAG method makes the use of the decision rules during the learning process. The decision trees can be constructed by a heuristically guided, hill climbing algorithm called ID3 ([12]). Its heuristic is based on an information-theoretic measure called *entropy*, which measures the length of the encoding of the current training set in bits. The most popular decision tree learner algorithm is the C4.5 system ([19]) which is widely used in academic and industrial spheres. There are many good textbooks available on decision tree learner methods ([12, 13, 19]). In the following, we represent a decision tree constructed for a learning task.

Example 2 *(A modified version of an example in [12].) The task is to find a concept which describes whether a robot is friendly or not, based on the properties Smiling, Holding, Has_tie, Head_Shape, Body_Shape, and an initial set of training examples.*

Smiling	*Holding*	*Has_tie*	*Head_Shape*	*Body_Shape*	*Class*
yes	*balloon*	*yes*	*square*	*square*	*friendly*
yes	*flag*	*yes*	*octagon*	*octagon*	*friendly*
yes	*balloon*	*no*	*round*	*round*	*friendly*
yes	*flag*	*no*	*octagon*	*octagon*	*friendly*
yes	*flag*	*no*	*octagon*	*octagon*	*friendly*
yes	*balloon*	*no*	*square*	*square*	*friendly*
yes	*sword*	*yes*	*round*	*octagon*	*unfriendly*
yes	*sword*	*no*	*square*	*octagon*	*unfriendly*
no	*sword*	*no*	*octagon*	*round*	*unfriendly*
no	*flag*	*no*	*round*	*square*	*unfriendly*



$Rule_1$: *Holding* = *balloon* → *class* ***friendly***
$Rule_2$: *Smiling* = *yes* ∧ *Holding* = *flag* → *class* ***friendly***
$Rule_3$: *Smiling* = *no* → *class* ***unfriendly***
$Rule_4$: *Holding* = *sword* → *class* ***unfriendly***

Default class: ***friendly***

Figure 3: The decision tree and decision rules constructed by the C4.5

These learning methods which generally yield robust, reliable results are even able to handle noisy input data and continuous attributes. However, they have some drawbacks as well. In the attribute-value-based representation, variables cannot be used, hence these learning methods cannot deal with complex relations. Another disadvantage is the inability of use of background knowledge.
The above problem was bridged by the introduction of inductive logic programming (ILP, [12, 14]). The learning methods developed in the ILP framework employ first-order logic to represent the learning task, the training examples and background knowledge. The latter is used intensively in the learning process.
The ILP learning system called LINUS ([12]) combines the advantages of attribute-value learners and first-order-logic-based representation. The learning approach of the LINUS system can be summarized in three steps:

- It transforms the learning task into a propositional form.
- The transformed learning task is solved by using an appropriate propositional learner.
- The results of this propositional learner are converted back into a first-order logic form.

A similar learning method (see Figure 4) is used in the AGLEARN algorithm for inducing attribute grammars. However, the AGLEARN describes the learning task and background knowledge used with the help of an attribute grammar instead of a logic program.

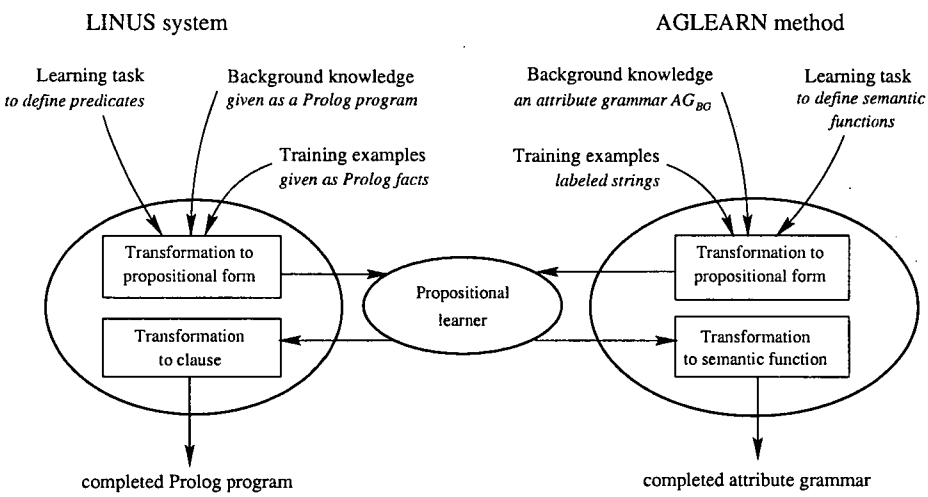


Figure 4: Similarities and differences between the LINUS system and the AGLEARN method

2.3 Description of the learning task

In this section, we formulate the learning task of *ags* in the following way:
The goal of the learning is to give a complete specification for the *ag* $AG = (G, SD, AD, R)$ from a *partially given L-attributed ag* $AG_{inp} = (G, SD, AD, R_{inp})$ and a set $\mathcal{W}_{inp}$ of *strings* taken from the language generated by AG_{inp}.
The term "partially given" here means that $R_{inp} \subseteq R$, namely some of the semantic functions of AG_{inp} are undefined. This AG_{inp} not only describes the background knowledge and the learning task, but is used to generate the training examples from the strings of $\mathcal{W}_{inp}$. The **background knowledge** is given as a fully defined *ag* $AG_{BG} = (G, SD_{BG}, AD_{BG}, R_{inp})$, where $SD_{BG} \subseteq SD$ and $AD_{BG} \subseteq AD$. The **learning task** is specified by the following items:

(1) The semantic domain $SD_{tar} = (\mathcal{T}_{tar}, \mathcal{F}_{tar})$ which consists the types of the target attributes ($\mathcal{T}_{tar}$) and initial functions ($\mathcal{F}_{tar}$) over these attributes. SD_{tar} is defined in advance, such that $SD_{tar} \cup SD_{BG} = SD$ holds. The LAG method constructs the unknown semantic functions from the elements of $\mathcal{F}_{tar}$.

(2) The description $AD_{tar} = (Inh_{tar}, Syn_{tar}, \tau)$ of the target attributes are related to the symbols of *cfg* G such that $AD_{tar} \cup AD_{BG} = AD$ holds.

(3) A set $TO(p)$ of the target attribute occurrences is assigned to production $p : X_0 \to X_1 \ldots X_{m_p}$ of G.
A defined attribute occurrence $X_l^p.a \in TO(p)\,(0 \leq l \leq m_p)$ if it has no semantic function in R_{inp}. In this case $X_l^p.a$ is called **target attribute occurrence**. $TO = \bigcup_p TO(p)$ denotes the set of all target attribute occurrences.

To be more exact, the learning method infers the unknown semantic functions of R_{tar} for the target attribute occurrences then completes the specification of AG_{inp} such that $R_{tar} \cup R_{inp} = R$ will hold.
The **training examples** for the target attribute occurrences are generated during the derivation of the input strings of $\mathcal{W}_{inp}$. Based on the AG_{inp}, a ddt_w is built for each $w \in \mathcal{W}_{inp}$ string. Let n_0 be a node of ddt_w labelled by X_0 and let $p : X_0 \to X_1 \ldots X_{m_p}$ be applied at this node. Moreover, let $X_1, \ldots, X_{m_p}$ each label the successor $n_1, \ldots, n_{m_p}$ of n_0, respectively. Then, during the traversal and evaluation of ddt_w for each instance $n_l.a$ of $X_l.a \in TO(p)$, $(0 \leq l \leq m_p)$, an example

$$e = (w, (u_1, v_1), \ldots (u_k, v_k), (n_l.a, v_0))$$

is added to the training set $\mathcal{E}_{X_l^p.a}$. The $v_1, \ldots, v_k$ denote the values of the instances of the used attribute occurrences $u_1, \ldots, u_k \in UO_{L-attr}(X_l^p.a)$ that have already been computed. With a knowledge of these values, the value v_0 of the target attribute instance $n_l.a$ is defined by the user.

Example 3 *We show what these definitions look like with the help of the type-checking example* AG_{typ} *(see Example 1). Let us suppose that the semantic functions in the production 1 and 3 are unknown:* $R(1) = R(3) = \emptyset$.

$$
\begin{aligned}
\textit{input strings } \mathcal{W}_{inp} = \{\ & ((3*2+6)-7)/(3*1.5-2.5/5),\\
& (3/2-1)*3+(0.7*(0.1+1))/(6*2+4.3)\}
\end{aligned}
$$

$$
\begin{aligned}
\textit{background knowledge } AG_{BG} &= (G_{typ}, SD_{typ}, AD_{typ}, R_{BG}),\ \textit{where } R_{BG} \subseteq R_{typ}\\
\textit{learning task } SD_{tar}\ \ \mathcal{T}_{tar} &= \{\{true, false\}\}\\
\mathcal{F}_{tar} &= \{=^2\},\ \textit{where } =^2 \textit{ is the identity relation}\\
AD_{tar}\ \ Syn &= \{mode\}\\
Syn(Expr) &= Syn(Term) = \{mode\}\\
\tau(mode) &= \{int, real\}\\
\textit{target attribute occurrences}\quad TO &= \{Expr_0^1.mode, Term_0^3.mode\}
\end{aligned}
$$

3 Learning of L-attributed grammars

In this section we introduce the *LAG* system which infers semantic functions for L-attributed grammars. It takes a partially given *ag* AG_{inp} and a set $\mathcal{W}_{inp}$ of strings of the language generated by AG_{inp} as input. The term 'partially given' here means that AG_{inp} has some attribute occurrences which have no semantic function. During the learning process the LAG method infers these unknown semantic functions and adds them to AG_{inp} to complete its specification.
AG_{inp} describes the learning tasks and the background *ag* AG_{BG}. In addition, it is used to generate the training examples from the strings of $\mathcal{W}_{inp}$. For each string a ddt is constructed by AG_{inp}, which also consists of instances of target attribute occurrences. During the evaluation of the ddt the values of these target instances are determined by the user with the knowledge of the values of other attribute instances. The latter have been computed automatically based on AG_{inp}. This is an important advantage of this system compared to other learning methods where a whole set of training examples have to be given in advance. After generating the training examples for the target attribute occurrences, the LAG system transforms the learning task and background knowledge into[2] an attribute-value representation.[3]
The learning tasks represented this way are solved by the decision tree learner, C4.5 ([19]). Finally, the hypotheses produced by the C4.5 in the form of decision rules are transformed back into *"if-then"* semantic functions (see Example 1).
The basic steps of the *LAG method* can be summarized as follows:

- *Generation of the training examples* from the input strings.
- *Transformation into attribute-value tuples*: a training table consisting of attribute-value tuples is constructed for each target attribute occurrence.
- *Decision tree learning*: solving the transformed learning tasks using the C4.5 system: the decision rules are built based on the training tables.
- *Formulating semantic functions*: The rules inferred by C4.5 are transformed back into the form of semantic functions.

[2] described by an attribute grammar
[3] expressed as attribute-value vectors

3.1 Generation of training examples

Using the input *ag* AG_{inp} we build a ddt_w for each input string w in $\mathcal{W}_{inp}$. In these *ddt*s the target attribute occurrences may have arbitrarily many instances.

Let n_0 be a node of ddt_w where the production $p : X_0 \to X_1 \ldots X_{m_p}$ is applied and let $n_l.a$, $(0 \leq l \leq m_p)$ be the instance of the target attribute occurrence $X_l^p.a \in TO(p)$. Further, let $UI_{L-attr}(n_l, a)$ denote the set of used attribute instances $n_{k_1}.u_1, \ldots, n_{k_s}.u_s \in UI(n_0, p)$, which fulfill the L-attributed conditions (see p. 283): they were computed before the evaluation of target attribute instance $n_l.a$.

During the evaluation of the ddt_w, the user is asked about the values of the target attribute instances by substituting the unknown semantic functions with a **question IQ**:

```
proc  IQ (set : UI_{L-attr}, inst : target);
      begin write('The used attribute instances have the following values:');
            write(UI_{L-attr}, p );
            read(target);
      end;
```

In addition, replacing using the procedure *new_eval*() instead of the procedure *eval*() in the *tree_traversal*() (see p. 283) process yields examples which are added to the training set $\mathcal{E}_{X_l^p.a}$ for each instance of the target attribute occurrence $X_l^p.a$.

```
proc new_eval (set : DI, node : n);
   begin for each a ∈ DI do
             if a ∉ TO then eval(n.a);
             else begin
                  a := IQ(UI_{L-attr}(n, a), a);
                  add_example(w, UI_{L-attr}(n, a), a);
                  end;
   end;
```

During the evaluation, one example is generated for each instance of each target attribute occurrence in the ddt_w. Hence examples can be produced for different training example sets. Since the training set $\mathcal{E}_{X_l^p.a}$ may contain an example more than once, even a small number of input strings can induce numerous training examples: $|\mathcal{W}_{inp}| \leq \left| \bigcup_{TO} \mathcal{E}_{X_l^p.a} \right|$.

Example 4 *(Continuing from Example 3) The training example set* $\mathcal{E}_{Expr_0^1.mode}$ *is generated for the target attribute occurrence* $Expr_0^1.mode$ *of production 1. (A similar training set can be constructed for the target attribute occurrence* $Term_0^3.mode$ *as well.)*

Let $w_1 = \big((3 * 2 + 6) - 7\big)/(3 * 1.5 - 2.5/5)$ *and*
$w_2 = (3/2 - 1) * 3 + \big(0.7 * (0.1 + 1)\big)/(6 * 2 + 4.3)$

denote the two input strings. The production 1 *is applied three times in* ddt_{w_1}*, hence three examples are generated for* $Expr_0^1.mode$ *during the traversal of* ddt_{w_1}*. Similarly, the traversal of* ddt_{w_2} *produces four examples. It is easy to check that* $\mathcal{E}_{Expr_0^1.mode}$ *is the following after the evaluation of* ddt_{w_1} *and* ddt_{w_2}*:*

	UO_{L-attr}			
$w \in \mathcal{W}_{inp}$	$Expr_1.mode$	$Addop.op$	$Term.mode$	$Expr_0.mode$
w_1	*int*	*add*	*int*	*INT*
w_1	*int*	*sub*	*int*	*INT*
w_1	*real*	*sub*	*real*	*REAL*
w_2	*real*	*sub*	*int*	*REAL*
w_2	*real*	*add*	*int*	*REAL*
w_2	*int*	*add*	*real*	*REAL*
w_2	*real*	*add*	*real*	*REAL*

3.2 Transformation into attribute-value tuples

Upon generating the training example set, the LAG method transforms the learning task into attribute-value tuples. One training table is generated for each target attribute occurrence (i.e. in the type-checking example two training tables are constructed: one for $Expr_0^1.mode$ and one for $Term_0^3.mode$).

There are two ways of formulating the training tables depending on the type of target attribute occurrences:

(1) Enumerated case: when the domain of the target attribute occurrence $X_l^p.a$ is defined by a finite list. In this case our aim is to infer a classification-like semantic function for it, where the classes are made up of the $c_1, \ldots, c_k$ elements of the domain. The training table $T_{X_l^p.a}$ consists of columns

$$\{string\} \cup UO_{L-attr} \cup \mathcal{R}_{\mathcal{U}} \cup \{class\},$$

where columns *string*, *class* and UO_{L-attr} are constructed from the training example set $\mathcal{E}_{X_l^p.a}$. The column *class* contains the value of $X_l^p.a$ computed during the evaluation of ddt_w, where $w \in \{string\}$. The set $\mathcal{R}_{\mathcal{U}}$ consists of the *satisfiable interpretations* of each relation $r : \tau(x_1) \times \cdots \times \tau(x_m) \rightarrow \{true, false\}$ given in SD_{inp}. An interpretation $r(u_1, \ldots, u_m)$ is satisfiable iff $u_i \in UO_{L-attr}$ and $\tau(u_i) = \tau(x_i)$, for all $i = 1 \ldots m$.

Example 5 *(Continuing from Example 4) Since we have only one relation '=' in SD_{inp}, the set $\mathcal{R}_{\mathcal{U}}$ only consists of the column r_1 : ($Expr_1.mode = Term.mode$). The training table $T_{Expr_0^1.mode}$ generated is:*

string	UO_{L-attr}			$\mathcal{R}_{\mathcal{U}}$	*class*
$w \in \mathcal{W}_{inp}$	$Expr_1.mode$	$Addop.op$	$Term.mode$	r_1	$Expr_0.mode$
w_1	*int*	*add*	*int*	*true*	*INT*
w_1	*int*	*sub*	*int*	*true*	*INT*
w_1	*real*	*sub*	*real*	*true*	*REAL*
w_2	*real*	*sub*	*int*	*false*	*REAL*
w_2	*real*	*add*	*int*	*false*	*REAL*
w_2	*int*	*add*	*real*	*false*	*REAL*
w_2	*real*	*add*	*real*	*true*	*REAL*

Based on the training tables constructed in this way the semantic functions are produced in the following form:

$$X_l^p.a = \begin{array}{l} \textbf{if } Test_{1,1} \wedge \cdots \wedge Test_{1,i_1} \textbf{ then } c_{i_1} \\ \textbf{else if } Test_{2,1} \wedge \cdots \wedge Test_{2,i_2} \textbf{ then } c_{i_2} \\ \quad \vdots \\ \textbf{otherwise } c_{i_s} \end{array}$$

where $c_{i_1}, \ldots, c_{i_s} \in \tau(X_l^p.a)$, and $Test_{k,j} (k = 1 \ldots s, j = i_1 \ldots i_s)$ is given in the form $(Column_{k,j} = v_j)$ with $Column_{k,j} \in UO_{L-attr} \cup \mathcal{R}_{\mathcal{U}}$ and $v_j \in \tau(Column_{k,j})$.

(2) Non-enumerated case: if the domain of a target attribute occurrence $X_l^p.a$ is non-enumerated then the LAG system is going to infer a semantic function for it by employing the elements of $\mathcal{F}_{tar}$. If so, a slightly extended training table $T_{X_l^p.a}$ is produced:

$$\{string, target\} \cup UO_{L-attr} \cup \mathcal{R}_{\mathcal{U}} \cup \mathcal{R}_{\mathcal{F}} \cup \{class\}.$$

The columns of the $string, UO_{L-attr}$, and $\mathcal{R}_{\mathcal{U}}$ are the same as those of the enumerated-typed target attribute occurrences. The main differences between the two cases surface in the columns of $\mathcal{R}_{\mathcal{F}}$, *target* and *class*.

The elements of the set $\mathcal{R}_{\mathcal{F}}$ are defined as a relation $(X_l^p.a = q)$, where q might be an attribute occurrence $u_k \in UO_{L-attr}$ or a satisfiable interpretation $f(u_1, \ldots, u_m)$ of $f : \tau(x_1) \times \cdots \times \tau(x_m) \rightarrow \tau(X_l^p.a)$. The values of $X_l^p.a$ computed during the parsing of the input strings make up the elements of the column *target*.

In addition, the elements of the column *class* $= \{+, -\}$ denote positive and negative examples. The positive examples of the training table will be elements of the set $\mathcal{E}_{X_l^p.a}$. The negative examples are generated from the positive ones by changing the elements of the column *target* with some randomly selected values of $\tau(X_l^p.a)$.

Example 6 *Let us consider an S-attributed ag AG_{ab}, which counts the number of letters in a string of the language a^*b^*.*

1, $S \rightarrow A\,B$	$S_0.n = A_1.n + B_2.n$	2, $A \rightarrow a\,A$	$A_0.n = inc(A_1.n)$
3, $A \rightarrow \lambda$	$A_0.n = 0$	4, $B \rightarrow b\,B$	$B_0.n = inc(B_1.n)$
5, $B \rightarrow \lambda$	$B_0.n = 0$		

Let us suppose that all of the semantic functions are unknown. The learning task will then defined as follows:

$$\begin{array}{ll} SD_{tar}: & \mathcal{T}_{tar} = \{\mathbb{N}\} \\ & \mathcal{F}_{tar} = \{inc^1, dec^1, +^2, -^2\}, \textit{ where} \\ & \quad inc: \mathbb{N} \rightarrow \mathbb{N} \qquad + : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \\ & \quad dec: \mathbb{N} \rightarrow \mathbb{N} \qquad - : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \\ AD_{tar}: & Syn = (n) \\ & Syn(S) = Syn(A) = Syn(B) = \{n\} \\ & \tau(n) = \mathbb{N} \\ \mathcal{W}_{inp} = & \{ab, aab, abb\} \\ TO = & \{S_0^1.n,\ A_0^2.n,\ A_0^3.n,\ B_0^4.n,\ B_0^5.n\} \end{array}$$

The training table $T_{A_0^2.n}$ for the target attribute occurrence $A_0^2.n$ consists of the following columns:

				$\mathcal{R}_\mathcal{F}$			
	$w \in \mathcal{W}_{inp}$	$A_0^2.n$	$A_1^2.n$	r_1	r_2	r_3	*class*
$UO_{L-attr} = \{A_1^2.n\}$	*ab*	1	0	*false*	*true*	*false*	+
$\mathcal{R}_\mathcal{U} = \emptyset$	*aab*	1	0	*false*	*true*	*false*	+
	aab	2	1	*false*	*true*	*false*	+
$r_1 : (A_0^2.n = A_1^2.n)$	*abb*	1	0	*false*	*true*	*false*	+
$r_2 : (A_0^2.n = inc(A_1^2.n))$	*ab*	2	0	*false*	*false*	*false*	−
$r_3 : (A_0^2.n = dec(A_1^2.n))$	*aab*	0	0	*true*	*false*	*false*	−
	aab	0	1	*false*	*false*	*true*	−
	abb	3	0	*false*	*false*	*false*	−

Similar training tables are generated for the target attribute occurrences $S_0^1.n$, $A_0^3.n$, $B_0^4.n$, $B_0^5.n$ as well.

Using the training tables structured in this way, the LAG system infers semantic functions which have the following form:

$$\begin{aligned} X_l^p.a = \; & \textbf{\textit{if}} \;\; Test_{1,1} \wedge \cdots \wedge Test_{1,i_1} \;\; \textbf{\textit{then}} \;\; q_{i_1} \\ & \textbf{\textit{else if}} \;\; Test_{2,1} \wedge \cdots \wedge Test_{2,i_2} \;\; \textbf{\textit{then}} \;\; q_{i_2} \\ & \quad \vdots \\ & \textbf{\textit{then}} \;\; q_{i_n} \end{aligned}$$

where $Test_{k,j}$ denotes the test $(Column_{k,j} = v_j)$ with $v_j \in \tau(Column_{k,j})$ and $Column_{k,j} \in UO_{L-attr} \cup \mathcal{R}_\mathcal{U}$. Here, q_{i_k} might be a function $f \in \mathcal{F}_{tar}$ or a used attribute occurrence $u \in UO_{L-attr}$.

3.3 Learning with the C4.5 system

The C4.5 system views the learning task described by the training table as a classification problem. The possible values of the target attribute occurrence make up the set of possible classes. The system constructs a classification model in the form of a decision tree or a set of decision rules. The LAG system formulates the semantic functions based on the decision rules.
The decision rules produced by the C4.5 system are represented as follows:

$$Rules_{X_l^p.a} = \left\{ \begin{array}{lll} Rule_1: & Column_{1,1} = v_1 & Column_{1,1} \in UO_{L-attr} \cup \mathcal{R}_\mathcal{U} \\ & \vdots & \\ & Column_{n_1} = v_{n_1} & Column_{n_1} \in UO_{L-attr} \cup \mathcal{R}_\mathcal{U} \\ & \rightarrow class \quad c_1 & c_1 \in \tau(X_l^p.a) \\ Rule_2: & \ldots & \\ \text{Default class:} & c_{default} & c_{default} \in \tau(X_l^p.a) \end{array} \right.$$

Example 7 *Based on the training table $T_{Expr_1^0.mode}$ given in the Example 5, the C4.5 system infers the following decision rules:*

$$Rules_{Expr_1^0.mode} = \left\{ \begin{array}{l} Rule_1: \;\; Expr_1.mode = real \rightarrow \;\; class\ REAL \\ Rule_2: \;\; Expr_1.mode = int \wedge \\ \qquad\quad Term.mode = int \rightarrow \;\; class\ INT \\ \\ Default\ class:\ REAL \end{array} \right.$$

Similar decision rules are inferred from the training table $T_{A_0^2.n}$ of the non-enumerated target attribute occurrence:

$$Rules_{A_0^2.n} = \begin{cases} Rule_1: & \big(A_0.n = inc(A_1.n)\big) = true & \rightarrow & class + \\ Rule_2: & \big(A_0.n = inc(A_1.n)\big) = false & \rightarrow & class - \\ Default\ class{:}+ & & & \end{cases}$$

3.4 Formulating semantic functions

First we simplify the set of rules learned by the C4.5 system then transform them into semantic functions.

(1) Enumerated case: The set of rules is reduced as follows:

$$Simplified\_Rules_{X_l^p.a} = \{r \in Rules_{X_l^p.a} \mid c_i \neq c_{default}\}.$$

This set is transformed to a semantic function of the form:

$$\begin{aligned} X_l^p.a = \ & \textbf{if}\ (Column_{1,1} = v_{i_1}) \wedge \cdots \wedge (Column_{1,n_1} = v_{1,n_1}) \\ & \textbf{then}\ c_1 \\ & \textbf{else}\ \textbf{if}\ (Column_{2,1} = v_{2,1}) \wedge \ldots \\ & \qquad \vdots \\ & \qquad \textbf{otherwise}\ c_{default} \end{aligned}$$

where $(Column_{i,j} = v_{i,j})$ occurs in the tests of $Simplified\_Rules_{X_l^p.a}$.

Example 8 *The semantic function formulated for the target attribute occurrence $Expr_0^1.mode$ is the following:*

$$\begin{aligned} Expr_0.mode = \ & \textbf{if}\ \ (Expr_1.mode = int) \wedge (Term.mode = int) \\ & \textbf{then}\ INT \\ & \textbf{else}\ REAL \end{aligned}$$

(2) Non-enumerated case: here, the rules inferred by C4.5 classify the examples into one of two classes: $+, -$. A rule is accepted iff it tests exactly one column of $\mathcal{R}_{\mathcal{F}}$.

The set $Simplified\_Rules_{X_l^p.a}$ is constructed in the following way:

$$Simplified\_Rules_{X_l^p.a} = \left\{ r_i \in Rules_{X_l^p.a} \;\middle|\; \begin{array}{c} (c_i = +)\,,\ and\ for\ exactly\ one\ k: \\ Column_{i,k} = (X_l^p.a = q_i) \in \mathcal{R}_{\mathcal{F}} \\ with\ (Column_{i,k} = true) \end{array} \right\}$$

This set is transformed to a semantic function in the form:

$$\begin{aligned} X_l^p.a = \ & \textbf{if}\ (Column_{1,1} = v_{i_1}) \wedge \ldots (Column_{1,n_1} = v_{1,n_1}) \\ & \textbf{then}\ q_1 \\ & \textbf{else}\ \textbf{if}\ (Column_{2,1} = v_{2,1}) \wedge \ldots \\ & \qquad \vdots \\ & \qquad \textbf{then}\ q_n \\ & \qquad \textbf{otherwise}\ \text{WARNING} \end{aligned}$$

where $Column_{i,j}$ are the tests of $Simplified\_Rules_{X_l^p.a}$, such that $Column_{i,j} \in UO_{L-attr} \cup \mathcal{R}_{\mathcal{U}}$, while q_i is a function and $(X_l^p.a = q_i)$ is

among the tests of $Simplified\_Rules_{X_l^p.a}$. (Note: if during the execution of the generated *ag* for a given input none of the conditions in the above semantic function are fulfilled, a warning message is induced for the user. This message indicates that the inferred semantic function is not applicable for that input. If the $Simplified\_Rules_{X_l^p.a} = \emptyset$, then it then means that the LAG system was not able to learn semantic function for $X_l^p.a$.)

Example 9 *The decision rules for the target attribute occurrence $A_0^2.n$ are simplified in the following way:*

$$Simplified\_Rules_{A_0^2.n} = \{Rule_1 : \ (A_0.n = inc(A_1.n)) = true \rightarrow \ class +\}$$

Since the simplified set of rules consists of a single rule not containing any tests over the elements of columns in $UO_{L-attr} \cup \mathcal{R}_{\mathcal{U}}$ and the test of this rule is an element of $\mathcal{R}_{\mathcal{F}}$, the generated semantic function of $A_0^2.n$ is

$$A_0.n = inc(A_1.n)$$

which is the correct solution.

Within the non-enumerated learning there is a special case where a constant value should be assigned to the target attribute occurrence. In this case a semantic function

$$X_l^p.a = c, \text{ where } c \in \tau(X_l^p.a)$$

is generated automatically based on a preliminary check of positive examples.

4 Application of the LAG method in NLP

4.1 Part-of-Speech Tagging Problem

Research into both text and spoken language understanding is significantly helped by investigating those phenomena that occur in actual language use.
The first stage of the investigation is to assign part of speech (PoS) tags to every word representing its syntactic category and morphological properties based on large corpora. The **corpus** is an archive of annotated words including their morphological properties as codes called **tag**. Annotating a given text is a far from trivial task since the words often belong to several syntactic categories or morphological classes in different contexts (e.g. the Hungarian word "*múlt*"[4] might be annotated by a verbal, noun or adjectival tag).
The task of a PoS tagger (morphological disambiguater) is to automatically select the appropriate PoS annotation in a given context where possible. In principle there are two main approach for automatic part-of-speech tagging:

- the probabilistic one which normally uses Hidden Markov Models and
- the rule-based one which normally uses linguistic rules.

[4] *múlt (verb) – passed (Perfect 'pass')*
múlt (noun) – past
múlt (adjective) – past, last

In this section we infer rules for a rule-based tagger with the aid of the LAG method. We specify an *ag* which detects correspondences among the parts of the sentences such as predicate phrase and subject phrase. Using this structural information during the learning process, the LAG system produces disambiguater rules for each ambiguous class.

4.2 The initial data set

Our Hungarian corpus is the morphologically annotated translation of George Orwell's novel *1984*. The first tagged version of this corpus was produced by the MULTEXT-East project ([7]). The corpus includes approximately 100 000 words including punctuation characters. The novel consists of four chapters where the first two served as training data for the learning process while chapters 3 and 4 were used as test data.
The most widely used encoding is the Morpho-Syntactical Description (MSD, [7]). Unfortunately it associates too many different classes with the Hungarian language. E.g. based on its stems, a *noun* could be annotated with 1324 different MSD codes. In order to reduce the number of MSD classes the CTAG encoding scheme (Corpus Tagging,[16]) was employed, which has just 120 word tags, 4 punctuation tags and 1 tag for *unknown* words. Table 2 lists the distribution of the ambiguous classes whose instances occur over 100 times in the training and test data.

Table 2: The most frequently ambiguous classes and their cardinalities

Classes	Occurrence		Classes	Occurrence	
	Training data	Test data		Training data	Test data
asn,vmis3s	490	182	nsn,rgn,rp	112	46
cp,rg	880	294	psn,rp	142	57
cp,rg,vmip3s	247	125	psn,t	1867	620
cp,rp	334	149	pso,rg	217	85
cp,vmis3s	113	38	rg,rp	150	59
ms,t	751	222	rg,st	285	100
nsn,psn	111	52			

For instance, a word which belongs to the ambiguous class $[asn, vmis3s]$ could be annotated as a *nominative, singular adjective* or as a *verb in past tense, 3rd person singular*. In another ambiguous case, the $[psn, t]$ stands for the word '*az*', which could be annotated as a *singular pronoun, nominative case*[5] or as an *article*[6]. (A brief description of the corpus tags is given in the Appendix B.)
Besides the tags there is an identifier associated with every sentence which shows the location of a sentence in the original text, namely *Orwell, Hungarian translation, 1st chapter, 2nd section, 1st paragraph, 1st sentence* is

[5] 'az' – the
[6] 'az' – that

'0hu.1.2.1.1',"*Derült*, hideg áprilisi nap volt,
az órák éppen tizenhármat ütöttek." [7]

This sentence is annotated as follows:

'0hu.1.2.1.1',(*asn*, [*asn*, *vmis3s*]), wpunct, asn, asn, nsn, vmis3s, wpunct,
(*t*, [*psn*, *t*]), npn, rg, msa, vmis3p, spunct

In the sequence of corpus tags an ambiguous case is denoted by a pair given in round brackets. The second component is the set of possible tags of the word, while the first component shows its correct tag in the given sentence. Using the sequences of corpus tags during the learning process we can infer context rules which describe general regularities among the morpho-syntactical categories of the language.
Each ambiguous class is dealt with as an independent learning task so we generate an initial input set for each one, based on sequences of the corpus tags. Each element of these input sets is structured as follows:

Sentence_ID, before$_1$, ..., before$_7$, after$_1$, ..., after$_7$, *correct_ctag*

where *correct_ctag* denotes the observed morpho-syntactical category of the word in the given sentence. In addition, we consider 7 corpus tags *before* and *after* the ambiguous case. (Here: we denote the blanks with *xxx* when this 7-sized window extends over the beginning and the end of a sentence).
Continuing our example, the following tuples are added to the input set $\mathcal{W}_{asn,vmis3s}$ and $\mathcal{W}_{psn,t}$ of the ambiguous class $[asn, vmis3s]$ and $[psn, t]$, respectively:

'0hu.1.2.1.1', xxx, xxx, xxx, xxx, xxx, xxx, xxx,
wpunct, asn, asn, nsn, vmis3s, wpunct, t, *asn*

'0hu.1.2.1.1', wpunct, vmis3s, nsn, asn, asn, wpunct, asn,
npn, rg, msa, vmis3p, spunct , xxx, xxx, *t*

Using these sets of sequences the C4.5 system can infer disambiguater rules for each ambiguous class, i.e. produce a set of decision rules for the class $[asn, vmis3s]$ such that:

Rule1: $before_1 = t \rightarrow$ *class* ***asn***
Rule 2: $after_1 = npn \rightarrow$ *class* ***asn***

⋮

Rule 36: $after_1 = spunct \rightarrow$ *class* ***vmis3s***
Rule 37: $before_1 = nsa \rightarrow$ *class* ***vmis3s***

Default class: ***vmis3s***

In order to generate more effective rules the LAG method has been designed to recognize structural coherences in the sentences via an *ag* and extend the input of C4.5 with them.

4.3 Description of the learning task

The *ag* AG_{ctag} introduced here, detects parts of sentences and phrases in ambiguous cases.
The parts of sentences can be derived from the corpus tags, which refer to the suffixes of the words as well. The suffix determines the role of a word in a sentence.

[7] It was a bright, cold day in April and the clocks were striking thirteen.

We separate the corpus tags into *groups* based on the role they play in a sentence such as *predicate, subject, object, attribute, dative adverb, other adverb.* The rest of the sentence elements are denoted with the value *other*. Furthermore, the value *none* is generated for the case of *xxx* tags.

The phrases, called *syntagmas*, describe relations among the parts of sentences like the *predicate syntagma*, where the predicate and subject are related, or the *accusative syntagma*, where the predicate and object are related. It is clear that the identification of a *syntagma* depends on the attribute *group*.

Furthermore, our experiments show that in most cases the choice of the correct corpus tag of a word is influenced only by its neighboring tags. Hence, we use a simplified *ag* AG_{ctag} as background knowledge which deals only with tags next to the ambiguous case (size of window = 1) and it detects a syntagma among the tags after it. (A part of the ag can be found in the Appendix C.)

G_{ctag} 1 : $Ctag\_Sentence \rightarrow$
Sentence_ID "," BeforeCtags "," AfterCtags Ctag_Sentence
2 : $Ctag\_Sentence \rightarrow \lambda$
⋮

SD_{ctag} $\mathcal{T}_{ctag} = \left\{ \begin{array}{l} \text{CTAG} = \{asn, asnx, \ldots, wmis3s, spunct, wpunct \ldots\} \\ \text{GROUP} = \{Pred, Subj, Acc, AdvDat, AdvOth, Att, Other, None\} \\ \text{SYNTAGMA} = \{PredSynt, SubjSynt, AccSynt, AdvDatSynt, \\ \qquad AdvOthSynt, AttSynt, OtherSynt, NoneSynt\} \end{array} \right\}$
$\mathcal{F}_{ctag} = \{=^2\}$ where, $=$ is the identity relation

AD_{ctag} $Inh = \emptyset$
$Syn = \{ctag_1, group_1, syntagma\}$
$Syn(BeforeCtags) = \{ctag_1, group_1\}$
$Syn(AfterCtags) = \{ctag_1, group_1, syntagma\}$

$\tau(ctag_1) = \text{CTAG}$
$\tau(group_1) = \text{GROUP}$
$\tau(syntagma) = \text{SYNTAGMA}$

In order to choose the contextually correct tag in an ambiguous case, a synthesized attribute *correct_ctag* is associated with the start symbol *Ctag_Sentence*. Its semantic function is unknown, so the learning task is described as follows:

the semantic domain SD_{tar}	$\mathcal{T}_{tar} = \{\text{CTAG}\}$
	$\mathcal{F}_{tar} = \emptyset$
the attribute description AD_{tar}	$Syn = \{correct\_ctag\}$
	$Syn(Ctag\_Sentence) = \{correct\_ctag\}$
	$\tau(correct\_ctag) = \text{CTAG}$
R_{tar}	$R(1) = \emptyset$
target attribute occurrence	$TO(1) = \{Ctag\_Sentence_0^1.correct\_ctag\}$
input strings i.e.	$\mathcal{W}_{inp} = \mathcal{W}_{asn,vmis3s}$

The learning concept is inferred by the LAG method introduced in the Section 3.

4.4 Generation of the training examples

We build the ddt_s for every given sequence s of corpus tags for an ambiguous class. Recalling that the values of the target attribute occurrence *correct_ctag* are defined in advance in the training corpus, the question IQ is not used during the tree traversals.
For instance, in the case of the ambiguous class $(asn, vmis3s)$ given the set of input sequences of $\mathcal{W}_{asn,vmis3s}$:

```
Ohu.1.2.1.1,   xxx, xxx, xxx, xxx, xxx, xxx, xxx
               wpunct, asn, asn, nsn, vmis3s, wpunct, t, asn
Ohu.1.2.5.5,   t, cp, wpunct, vmn, vmis3s, rg, i, nso
               rg, vmip3p, rq, t, nsa, spunct, asn
...
Ohu.2.11.40.5, rg, rg, spunct, rp, vmcp3s, nsax, cp
               pso, ms, nsa, spunct, xxx, xxx, xxx, vmis3s
Ohu.2.11.40.5, nsa, asn, asn, xxx, xxx, xxx, xxx
               nso, wpunct, cp, nsax, vmcp3s, rp, spunct, vmis3s
```

The training example set $\mathcal{E}_{asn,vmis3s}$ generated in this case is:

Sentence_ID	UO_{L-attr}					*class*
	u_1	u_2	u_3	u_4	u_5	
Ohu.1.2.1.1	xxx	none	wpunct	Oth	AttSynt	*asn*
Ohu.1.2.5.5	t	Oth	nso	AdvOth	AdvOthSynt	*asn*
...						
Ohu.2.11.40.5	rg	Oth	pso	AdvOth	noneSynt	*vmis3s*
Ohu.2.11.40.5	nsa	Acc	nso	AdvOth	AdvOthSynt	*vmis3s*

$u_1 : BeforeCtags.ctag_1$ $u_2 : AfterCtags.ctag_1$
$u_3 : BeforeCtags.group_1$ $u_4 : AfterCtags.group_1$
$u_5 : AfterCtags.syntagma$

$class : Ctag\_Sentences.correct\_tag$.

4.5 Preparation of the training tables

Since the target attribute occurrence *Ctag_Sentences.correct_tag* is enumerated-typed, the training table consists of the columns

$\{Sentence\_ID\} \cup UO_{L-attr} \cup \mathcal{R}_{\mathcal{U}} \cup \{correct\_ctag\}$

where $\mathcal{R}_{\mathcal{U}}$ contains the relations

r_1 : *(BeforeCtags.ctag*$_1$ = *AfterCtags.ctag*$_1$*)*
r_2 : *(BeforeCtags.group*$_1$ = *AfterCtags.group*$_1$*)*

Hence, the training table $T_{asn,wmis3s}$ is constructed as follows:

Sentence_ID	UO_{L-attr}					$\mathcal{R}_{\mathcal{U}}$		*class*
	u_1	u_2	u_3	u_4	u_5	r_1	r_2	
Ohu.1.2.1.1	xxx	None	wpunct	Oth	AttSynt	false	false	*asn*
Ohu.1.2.5.5	t	Oth	nso	AdvOth	AdvOthSynt	false	false	*asn*
...								
Ohu.2.11.40.5	rg	Oth	pso	AdvOth	noneSynt	false	false	*vmis3s*
Ohu.2.11.40.5	nsa	Acc	nso	AdvOth	AdvOthSynt	false	false	*vmis3s*

4.6 Inferred context rules

The sets of decision rules are inferred based on the training tables, i.e.:

$$Rules_{asn,vmis3s}\begin{cases} Rule_1: & BeforeCtags.group_1 = Acc & \rightarrow class\ vmis3s \\ Rule_2: & AfterCtags.group_1 = Oth & \rightarrow class\ vmis3s \\ Rule_3: & AfterCtags.ctag_1 = pso & \\ & AfterCtags.syntagma = AttSynt & \rightarrow class\ asn \\ \vdots & & \\ Rule_{42}: & BeforeCtags.ctag_1 = wpunct & \\ & AfterCtags.syntagma = AttSynt & \rightarrow class\ asn \\ Default\ class:\ vmis3s & & \end{cases}$$

The rule sets are reduced and converted to the form of semantic functions. Let us take for instance the case of the ambiguous class *asn, vmis3s*:

$$\begin{aligned} Ctags\_Sentences.correct\_tag = \textbf{if}\ & (BeforeCtags.ctag_1 = wpunct)\ and \\ & (AfterCtags.syntagma = AttSynt) \\ \textbf{then}\ & asn \\ \textbf{else}\ & \textbf{if}\ (AfterCtags.ctag_1 = pso)\ and \\ & (AfterCtags.syntagma = AttSynt) \\ & \textbf{then} \\ & \ldots \\ & \textbf{else}\ vmis3s \end{aligned}$$

Since disambiguater rules for any ambiguity can be inferred this way the above method is a useful tool for a PoS tagger system.

5 Comparison of the results of C4.5 and LAG

In the following table we compare the accuracy of the disambiguater rules achieved by C4.5 and LAG based on the corpus of Orwell's novel. The accuracy of the rules is tested using the chapters 3 and 4 of the novel, these chapters not being used during training process.

Table 3 shows the error numbers and error percentages of the decision rule sets generated for the most frequent ambiguous classes. The rules inferred by the C4.5 system are based on the sequences of corpus tags (see p. 297). The LAG system, however, creates its results by the means of the training sequences which are augmented with structural information detected by the *ag* given in Section 4.3 In the column *Mark*, the sign

"+" denotes those classes where the use of LAG yields only minor improvements, and

"++" means significant improvements produced by employing the LAG method compared to C4.5.

The results show the accuracy of the inferred rules is improved if an *ag* as background knowledge is utilised during the learning process.

Table 3: *The comparison of the C4.5 and LAG system*

Ambiguity classes	Results by C4.5				Results by LAG				Mark
	training data		test data		training data		test data		
	#err	err %	#err	err %	#err	err %	#err	err %	
asn-vmis3s	39	8.0 %	15	8.2 %	34	6.9 %	11	6.0 %	+
cp-rg	142	16.1 %	72	24.5%	136	15.5 %	69	23.5%	+
cp-rg-vmip3s	14	5.7 %	31	24.8%	11	4.5 %	23	18.4%	+
cp-rp	41	12.3 %	16	10.7%	10	3.0 %	11	7.4 %	++
cp-vmis3s	2	1.8 %	0	0.0 %	0	0.0 %	0	0.0 %	+
nsn-psn	24	21.6 %	16	30.8%	4	3.6 %	6	11.5%	++
psn-rp	9	6.3 %	3	5.3 %	6	4.2 %	3	5.3 %	+
psn-t	28	1.5 %	17	2.7 %	25	1.3 %	15	2.4 %	+
pso-rg	73	33.6 %	34	40.0%	25	11.5 %	11	12.9%	++
rg-rp	57	38.0 %	15	25.4%	31	20.7 %	8	13.6%	++
rg-st	104	36.5 %	44	44.0%	62	21.8 %	35	35.0%	++

6 Summary

In this paper we investigated the specification of *ags* from the viewpoint of inductive learning. We described a learning task for inferring semantic functions of a partially defined *ag* and introduced an inductive learning method for solving this task. In the learning approach of the LAG system a number of similarities exist between it and ILP methods. These similarities arise from the close connection between logic programs and *ags*. The LAG method infers semantic functions for enumerated and non-enumerated attribute occurrences of an L-attributed or S-attributed grammar. During the learning process it derives the training examples from input strings with the help of background knowledge. The background knowledge given as an *ag* is employed in the preparation the training tables for the target attribute occurrences. Using the training tables the C4.5 system produces decision rules which are then converted to the form of semantic functions.

We plan to increase the efficiency of the LAG method by reducing the restrictions related to background knowledge, i.e. extend the the algorithm to more complex *ags* than the S-attributed and L-attributed ones. Moreover, we would like to develop a more precise algorithm for the non-enumerated cases.

As regards to the PoS tagging application we would also like improve the background attribute grammar to better describe the features of the Hungarian language.

References

[1] ALBLAS, H.: Introduction to Attribute Grammars. Springer Verlag LNCS 545, p.1–16, 1991.

[2] ALEXIN, Z., ZVADA, SZ., GYIMÓTHY, T.: Application of AGLEARN on Hungarian Part-of-Speech Tagging. In Proceedings of the Second Workshop on Attribute Grammars and their Applications (WAGA'99), Amsterdam, The Netherlands. p.133–152, INRIA Rocquencourt, 1999.

[3] CUSSENS, J.: Part-of-Speech Tagging Using Progol. In Proceedings of the Seventh International Workshop on Inductive Logic Programming (ILP97), Prague, Czech Republic, Springer Verlag LNAI 1297, p.37–44, 1997.

[4] DERANSART, P., JOURDAN, M., LORHO, B.: Attribute Grammars - Definitions, Systems and Bibliography. Springer Verlag LNCS 323, 1988.

[5] DERANSART, P., MALUSZYŃSKI, J.: Relating Logic Programs and Attribute Grammars. Journal of Logic Programming 2, p.119–156, 1985.

[6] DERANSART, P., MALUSZYŃSKI, J. : A Grammatical View of Logic Programming. MIT Press, 1993.

[7] ERJAVEC, T., MONACHINI, M.: Specification and Notation for Lexicon Encoding. Copernicus Project 106 "MULTEXT-EAST", Deliverable D1.1 F (Final Report). 1997

[8] GYIMÓTHY, T., HORVÁTH, T.: Learning Semantic Functions of Attribute Grammars. Nordic Journal of Computing 4, p.287–302, 1997.

[9] HORVÁTH, T., ALEXIN, Z., GYIMÓTHY, T., WROBEL, S.: Application of Different Learning Methods to Hungarian Part-of-speech Tagging. In Proceedings of Ninth Workshop on Inductive Logic Programming (ILP99), Bled, Slovenia, Springer Verlag LNAI 1634, p.128–139, 1999.

[10] KASTENS, U.: Ordered Attribute Grammars. Acta Informatica 13, p.229–256, 1980.

[11] KNUTH, D.E.: Semantics of Context-Free Languages. Mathematical Systems Theory 2(2), p.127–145, 1968. Correction: Mathematical Systems Theory 5(1), p.95–96, 1971.

[12] LAVRAČ, N., DŽEROSKI, S.: Inductive Logic Programming: Techniques and Applications. Ellis Horwood, New York, 1994.

[13] MITCHELL, T. Machine Learning. McGraw-Hill, 1997.

[14] MUGGLETON, S.: Inductive Logic Programming. Academic Press, London, 1992.

[15] MUGGLETON, S., DE RAEDT, L.: Inductive Logic Programming: Theory and Methods. Journal of Logic Programming 19/20, p.629–679, 1994.

[16] ORAVECZ, CS.: Part-of-Speech Tagging in the Hungarian National Corpus – a Case Study. Research Institute for Linguistics of the Hungarian Academy of Sciences, 1998.

[17] PAAKKI, J.: A Logic-Based Modification of Attribute Grammars for Practical Compiler Writing. In Proceedings of the Seventh International Conference on Logic Programming (D.H.D. Warren, P. Szeredi, eds.), Jerusalem, p.203–217. MIT Press, 1990.

[18] QUINLAN, J.R. Induction of decision trees. Machine Learning, 1(1), p.81–106, 1986.

[19] QUINLAN, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

A Appendix

The corpus tags used in the Hungarian translation of the Orwell novel '1984':

ASN, ASNX, ASNY, ASA, ASAX, ASAY, ASD, ASDX, ASDY, ASO, ASOX, ASOY, APN, APNX, APNY, APA, APAX, APAY, APD, APDX, APDY, APO, APOX, APOY, MP, MPX, MPY, MS, MSX, MSY, MD, I, CP, NSN, NSNX, NSNY, NSA, NSAX, NSAY, NSD, NSDX, NSDY, NSO, NSOX, NSOY, NPN, NPNX, NPNY, NPA, NPAX, NPAY, NPD, NPDX, NPDY, NPO, NPOX, NPOY, PSN, PSNX, PSNY, PSA, PSAX, PSAY, PSD, PSDX, PSDY, PSO, PSOX, PSOY, PN, PPNX,PPNY, PPA, PPAX, PPAY, PPD, PPDX, PPDY, PPO, PPOX, PPOY, RG, RO, RP,RQ, RV, ST, T, VA, VMCP1S, VMCP1P, VMCP2, VMCP2S, VMCP2P, VMCP3S, VMCP3P, VMIP1S, VMIP1P, VMIP2, VMIP2P, VMIP2S, VMIP3S, VMIP3P, VMIS1S, VMIS1P, VMIS2, VMIS2P, VMIS2S, VMIS3S, VMIS3P, VMMP1S, VMMP1P, VMMP2, VMMP2P,VMMP2S, VMMP3S, VMMP3P, VMN, CPUNCT, SPUNCT, WPUNCT, UNKNOWN, X,Y

B Appendix

Here we briefly describe the above mentioned corpus tags. The first letter of each *ctag* stands for the *category* of the related words:

Ctag	Category	Ctag	Category
A	Adjective	R	Adverb
CP	Conjuntion	ST	Postposition
I	Interjection	T	Article
M	Numeral	V	Verb
N	Noun	X	Residual
P	Pronoun	Y	Abbreviation
SPUNCT	sent. punct.	CPUNCT	closing punct.
WPUNCT	wordpunct.		

Then the tags are constructed in the following way:

After A, N, M and P : The second letter after A, N, M and P denotes the *cardinality* while the third one is related to the *cases*, and the fourth letter refers to the *possessive* cases:

Position 2	Position 3	Position 4
	N nominative	
S singular	A accusative	X/M.X
P plural	D dative	Y/M.Y
	O other	

After V : in the case of verbs the situation is the following:

Position 2	Position 3 modes	Position 4 tenses	Position 5 person	Position 6
	I indicative			
M main	M imperative	P present	1	S single
A auxiliary	C conditional	S past	2	P prural
	N infinitive		3	

Other combination : MD numeral digit
RG general adverb
RP verbal participle
RV present partiple
RQ interrogative clitic

C Appendix

A part of the background ag AG_{ctag} defined for PoS tagging problem is:

```
Ctags_Sentences → Sentence_ID "," BeforeCtags "," AfterCtags Sentences
Ctags_Sentences → λ
...
AfterCtags → Acc_Group "," Synt_Acc          syntagma = Synt_Acc.syntagma
                                             ctag = Acc_Group.ctag
                                             group = Acc

AfterCtags → Pred_Group "," Synt_Pred        syntagma = Synt_Pred.syntagma
                                             ctag = Pred_Group.ctag
                                             group = Pred
...
Synt_Acc → Pred_Group "," Ctags              syntagma= AccSynt
Synt_Acc → NonPred_Group "," Synt_Acc        syntagma= Synt_Acc.syntagma

Synt_AdvDat → Pred_Group "," Ctags           syntagma= AdvDatSynt
Synt_AdvDat → NonPred_Group "," Synt_AdvDat  syntagma= Synt_AdvDat.syntagma

Synt_Subj → Pred_Group "," Ctags             syntagma= SubjSynt
Synt_Subj → NonPred_Group "," Synt_Subj      syntagma= Synt_Subj.syntagma
...
Ctags → 'asn'
Ctags → 'asnx'
...
```

Acta Cybernetica **15** (2001) 305–320.

A Fuzzy Approach for Mining Quantitative Association Rules

Attila Gyenesei *

Abstract

During the last ten years, data mining, also known as knowledge discovery in databases, has established its position as a prominent and important research area. Mining association rules is one of the important research problems in data mining. Many algorithms have been proposed to find association rules in databases with quantitative attributes. The algorithms usually discretize the attribute domains into sharp intervals, and then apply simpler algorithms developed for boolean attributes. An example of a quantitative association rule might be "10% of married people between age 50 and 70 have at least 2 cars". Recently, fuzzy sets were suggested to represent intervals with non-sharp boundaries. Using the fuzzy concept, the above example could be rephrased e.g. "10% of married old people have several cars". However, if the fuzzy sets are not well chosen, anomalies may occur. In this paper we tackle this problem by introducing an additional fuzzy normalization process. Then we present the definition of quantitative association rules based on fuzzy set theory and propose a new algorithm for mining fuzzy association rules. The algorithm uses generalized definitions for interest measures. Experimental results show the efficiency of the algorithm for large databases.

1 Introduction

The goal of data mining is to extract higher-level information from an abundance of raw data. Mining association rules is one of the important research problems in data mining [11]. The problem of mining boolean association rules over basket data was introduced in [1]. Given a set of transactions where each transaction is a set of items, an association rule is an expression of the form $X \Rightarrow Y$, where X and Y are sets of items. An example of an association rule is: "40% of transactions that contain beer and potato chips also contain diapers; 5% of all transactions contain all of these items". Here 40% is called the confidence of the rule, and 5% the support of the rule. The problem is to find all association rules that satisfy user-specified minimum support and minimum confidence constraints. There are many known algorithms for mining boolean association rules (see [2], [4], [5], [10] and [13] for just a few examples).

*Turku Centre for Computer Science (TUCS), University of Turku, Department of Computer Science, Lemminkäisenkatu 14, FIN-20520 Turku, Finland, e-mail: gyenesei@cs.utu.fi

In practice the information in many, if not most, databases is not limited to categorical attributes (e.g. zip code, make of car), but also contains much quantitative data (e.g. age, income). The problem of mining quantitative association rules was introduced and an algorithm proposed in [12]. The algorithm involves discretizing the domains of quantitative attributes into intervals in order to reduce each domain into a categorical one. An example of such an association might be "10% of married people between 50 and 70 have at least 2 cars".

Without a priori knowledge, however, determining the right intervals can be a tricky and difficult task due to the "catch-22" situation, as called in [12], because of the effects of *small support* and *small confidence*. Moreover, these intervals may not be concise and meaningful enough for human experts to easily obtain nontrivial knowledge from those rules discovered.

Instead of using sharp intervals, fuzzy sets were suggested in [9] to represent intervals with non-sharp boundaries. The obtained rules are called fuzzy association rules. If meaningful linguistic terms are assigned to fuzzy sets, the fuzzy association rule is more understandable. The above example could be rephrased e.g. "10% of married old people have several cars". An algorithm for mining fuzzy association rules was proposed in [8], but the problem is that an expert must provide the required fuzzy sets of the quantitative attributes and their corresponding membership functions. It is unrealistic to assume that experts can always provide the best fuzzy sets for fuzzy association rule mining. Moreover, if the fuzzy sets are not well chosen, anomalies may occur. In this paper we will tackle this problem by introducing an additional fuzzy normalization process.

The rest of this paper is organized as follows. In the next section, we present a brief description of how existing algorithms can be used for the mining of quantitative association rules and how fuzzy techniques can be applied to the data mining process. Then we will introduce a fuzzy normalization process in Section 3. In the same section, we give the definitions of fuzzy association rules and interest measures. In Section 4 we propose a new algorithm for fuzzy quantitative association rules. In Section 5 the experimental results are reported, followed by a brief conclusion in Section 6.

2 Problem Description

Several efficient algorithms for mining boolean association rules have been presented. Boolean attributes can be considered a special case of categorical attributes [4] and it is relatively straightforward to generalize the boolean algorithms for categorical attributes. For quantitative attributes, however, the situation is not so simple. We either have to somehow transform the quantitative association rules problem into boolean one or to find new algorithms. Here we shall, in fact, apply both alternatives.

2.1 Mapping Quantitative Attributes to Boolean Ones

If the quantitative association rules problem can be mapped to the boolean association rules problem, any algorithm for finding boolean association rules can be used to find quantitative association rules. This mapping can be performed as follows [12]. Suppose that we have a database shown in Table 1.

RID	Age	Income	Status	RID	Age	Income	Status
1	19	1400	Unmarried	11	26	2000	Married
2	22	1600	Unmarried	12	31	2400	Married
3	31	2400	Unmarried	13	19	1400	Unmarried
4	18	1400	Married	14	27	2200	Married
5	23	1600	Married	15	31	2600	Married
6	30	2800	Married	16	15	1000	Unmarried
7	17	1200	Unmarried	17	24	1800	Unmarried
8	25	2000	Married	18	38	2600	Married
9	31	2200	Married	19	17	1200	Unmarried
10	19	1400	Unmarried	20	39	2400	Married

Table 1: An example database

Let the relational table contain a boolean field for each attribute value/interval for each quantitative attribute. Then the value of any such boolean field, which corresponds to $\langle attribute, v \rangle$, would be "1" if the *attribute* had v in the original record, and "0" otherwise. Table 2 shows this mapping for the example database given in Table 1. Age is partitioned into three intervals: 11..20, 21..30 and 31..40. For income, two intervals have been defined. The categorical attribute, Status, is represented by two boolean attributes: "Unmarried" and "Married".

RID	Age (11..20)	Age (21..30)	Age (31..40)	Income (1000..1800)	Income (2000..2800)	Status Unmarried	Status Married
1	1	0	0	1	0	1	0
2	0	1	0	1	0	1	0
3	0	0	1	0	1	1	0
4	1	0	0	1	0	0	1
5	0	1	0	1	0	0	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
19	1	0	0	1	0	1	0
20	0	0	1	0	1	0	1

Table 2: Mapping to boolean association rules problem

For example, Record 5, which had $\langle Age : 23 \rangle$ now has "Age : 11..20" equal to "0", "Age : 21..30" equal to "1", and Age : 31..40" equal to "0", etc.

2.2 Mapping Problems

Unfortunately, the mapping approach leads to two problems [12]:

- *Small support*: if an interval is too small, a rule containing this interval may not have the minimum support; either very few rules are generated or rules are nearly as specific as the data itself.
- *Small confidence*: if an interval is too large, a rule containing this interval in the antecedent may not have the minimum confidence; many rules with little information are generated.

An example of the problem of small support (also called "sharp boundary problem" in [9]) is shown in Figure 1, suppose $[11, 20]$, $[21, 30]$ and $[31, 40]$ are three intervals created on the quantitative attribute Age, with 35%, 35% and 30% supports. If the minimum support threshold is a bit greater than 35%, then none of the intervals has sufficient support. However, there are high frequencies at 19 and 31, so a small extension of the interval $[21, 30]$ would make it frequent.

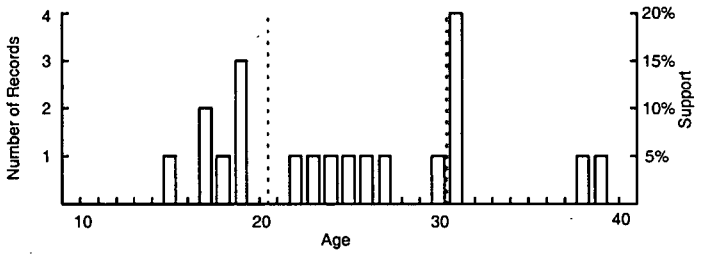


Figure 1: Example of small support problem

Of course, there is no restriction that the intervals should be disjoint. By letting them overlap, the sharp boundary problem can be overcome [12], see Figure 2.

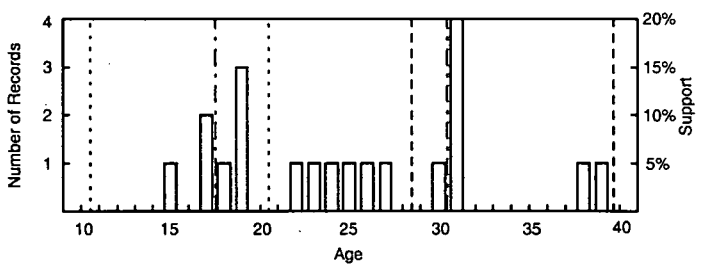


Figure 2: Overlapping adjacent intervals

Combining/overlapping adjacent intervals avoids the small support problem, and the number of intervals may be increased to avoid the small confidence problem. Unfortunately, this approach introduces a new problem [12]:

- *Many rules*: Consider an interval satisfying the minimum support. Then any range containing this interval will also satisfy the minimum support. Thus, the number of rules increases, and not all of them are interesting.

2.3 Fuzzy Approach

Instead of using sharp intervals, fuzzy sets were suggested in ([3], [9]) to represent intervals with non-sharp boundaries, as shown in Figure 3. Using fuzzy sets, an element can belong to a set with set membership value in [0, 1]. The lower histogram in Figure 3 shows membership values chosen for the middle fuzzy set.

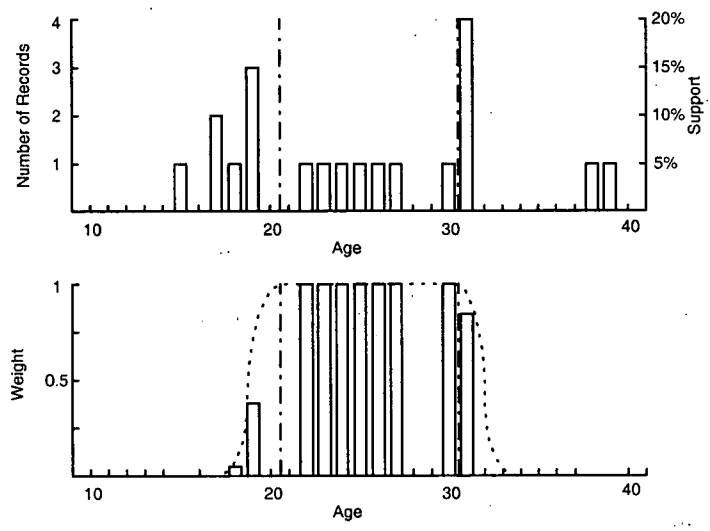


Figure 3: Fuzzy set

However, if the fuzzy sets are not well chosen, some anomalies occur. In Figure 1, the three intervals will be replaced by three fuzzy sets. Suppose the value 30 has membership degree of 0.9 in the second set and 0.3 in the third set. Then it will contribute 0.9 to the support of the second fuzzy set and 0.3 to the third one. However, this means that the value 30 will be more important than other values since the sum of its contributions to different fuzzy sets has become greater than 1. In the following section we will tackle this problem by introducing an additional fuzzy normalization process.

3 Inclusion of Fuzzyness in Associaton Rules

Our starting point is that the fuzzy sets and their membership functions are given. In [6] we gave a clustering algorithm for their automatic generation, but here we do not make any assumptions of the source of fuzzy sets. In this section we will first

introduce a fuzzy normalization process, to derive unbiased membership functions for the given fuzzy sets. Then we will give the generalized definition of a fuzzy association rule and related interest measures.

3.1 Fuzzy Normalization Process

Let $I = \{i_1, i_2, \ldots, i_n\}$ be the complete set of items where each i_j $(1 \leq j \leq n)$ denotes a categorical or quantitative (fuzzy) attribute. Further denote by $F(i_j) = \{\langle i_j, l\rangle \mid l = 1, \ldots, N(i_j)\}$ the set of fuzzy sets (or non-fuzzy categories), related to item i_j, where $N(i_j)$ represents the number of fuzzy sets (or number of categories). The membership function of $\langle i_j, l\rangle$ is denoted by $m_{\langle i_j,l\rangle}(v)$. If i_j is categorical, $m_{\langle i_j,l\rangle}(v) = 0$ or $m_{\langle i_j,l\rangle}(v) = 1$. If i_j is fuzzy, $0 \leq m_{\langle i_j,l\rangle}(v) \leq 1$. Thus, categories are special fuzzy sets, and can be handled similarly.

Let $t = \{t.i_1, t.i_2, \ldots, t.i_n\}$ be a transaction, where $t.i_j$, $(1 \leq j \leq n)$ represents the value of the j^{th} item. Value $t.i_j$ can be mapped to

$$\{(l, m_{\langle i_j,l\rangle}(t.i_j)) \mid \text{ for all } l, 1 \leq l \leq N(i_j)\}.$$

We define that $F(i_j)$ is a *'fuzzy partition'* if $\sum_{l=1}^{N(i_j)} m_{\langle i_j,l\rangle}(v) = 1$ for each v in domain i_j where i_j is fuzzy. This is a natural generalization to the non-fuzzy partitioning of a set into disjoint intervals covering the whole range. In practice, the sum may not always be equal to 1. We therefore define a normalization process as follows:

$$m'_{\langle i_j,l\rangle}(t.i_j) = \frac{m_{\langle i_j,l\rangle}(t.i_j)}{\sum_{l=1}^{N(i_j)} m_{\langle i_j,l\rangle}(t.i_j)}$$

Example. Suppose $I = \{status, age\}$ where *status* is a categorical attribute with the domain of {married, unmarried} and *age* is a quantitative attribute with three fuzzy sets {young, middle, old}. Note that it is possible to define other fuzzy set groups for this attribute. t = {unmarried, 25} will be mapped to {{(married, 0),(unmarried, 1)}, {(young, 0.2), (middle, 0.9), (old, 0.1)}}.

$(Status, married)$	$(Status, unmarried)$	$(Age, young)$	$(Age, middle)$	(Age, old)
0	1	0.2	0.9	0.1

Table 3: Without fuzzy normalization

Without normalization (Table 3), transaction t would increase the support of itemset $\{Status = unmarried, Age = young\}$ by 0.2, the support of itemset $\{Status = unmarried, Age = middle\}$ by 0.9, and the support of itemset $\{Status = unmarried, Age = old\}$ by 0.1. That is to say, this transaction will be counted $0.2 + 0.9 + 0.1 = 1.2$ times for the item *Age*. However, it is unreasonable for one transaction to contribute more than others, if the corresponding discrete sets are disjoint.

$(Status, married)$	$(Status, unmarried)$	$(Age, young)$	$(Age, middle)$	(Age, old)
0	1	0.167	0.75	0.083

Table 4: After fuzzy normalization

In contrast (Table 4), the normalization process will further transform the transaction t into {{(married, 0),(unmarried, 1)}, {(young, 0.167), (middle, 0.75), (old, 0.083)}}, for a total contribution of 1.0 for the item *Age*.

It should be noticed that normalization is not always order-preserving, with respect to the values of membership functions. It might even produce functions which are not concave. For example, suppose that we have a quatitative attribute *age* and three transactions $t_1 = \{25\}$, $t_2 = \{26\}$ and $t_3 = \{27\}$. Table 5 shows the original and normalized mappings of the transactions into membership values.

Transaction	*Age*	Original memberships $\{young, middle, old\}$	Normalized memberships $\{young, middle, old\}$
t_1	25	{0.20, 0.90, 0.10}	{0.167, 0.750, 0.083}
t_2	26	{0.20, 0.91, 0.11}	{0.164, 0.746, 0.090}
t_3	27	{0.18, 0.92, 0.12}	{0.148, 0.754, 0.098}

Table 5: Example of normalization anomaly

Notice the anomaly for the 'middle' fuzzy set: normalization changes the order of membership values. A sufficient (but not necessary) condition for concavity is that $\sum_{l=1}^{N(i_j)} m_{\langle i_j,l\rangle}(v)$ is constant for all v in domain i_j. In [6] we tackled this problem and showed how to create a fuzzy partition directly, without normalization.

3.2 Fuzzy Association Rule

After having obtained the fuzzy partitions and their corresponding membership functions for each fuzzy set of every quantitative attribute, a new transformed (fuzzy) database D^T is generated from the original database. Given a database $D^T = \{t_1, t_2, \ldots, t_n\}$ with attributes I and the fuzzy sets $F(i_j)$ associated with attributes i_j in I, we use the following form for a fuzzy association rule [9]:

$$\textbf{If } X = \{x_1, x_2, \ldots, x_p\} \textbf{ is } A = \{a_1, a_2, \ldots, a_p\}$$
$$\textbf{then } Y = \{y_1, y_2, \ldots, y_q\} \textbf{ is } B = \{b_1, b_2, \ldots, b_q\},$$

where $a_i \in F(x_i)$, $i = 1, \ldots, p$, and $b_j \in F(y_j)$, $j = 1, \ldots, q$. X and Y are ordered subsets of I and they are disjoint i.e. they share no common attributes. A and B contain the fuzzy sets associated with the corresponding attributes in

X and Y. As in the binary association rule, "X is A" is called the *antecedent* of the rule while "Y is B" is called the *consequent* of the rule. We also denote $Z = X \cup Y = \{z_1, \ldots, z_{p+q}\}$ and $C = A \cup B = \{c_1, \ldots, c_{p+q}\}$.

3.3 Fuzzy Itemset Measures - Support and Confidence

Let $D^T = \{t_1, t_2, \ldots, t_n\}$ be a database, where n denotes the total number of records ('transactions'). Let (Z, C) be an attribute-fuzzy set pair, where Z is an ordered set of attributes z_j and C is a corresponding set of fuzzy sets c_j. (From now on, we prefer to use the word "itemset" instead of "attribute-fuzzy set pair" for (Z, C) elements). If a fuzzy association rule $(X, A) \rightarrow (Y, B)$ is interesting, it should have enough fuzzy support $FS_{(Z,C)}$ and a high fuzzy confidence value $FC_{((X,A),(Y,B))}$, where $Z = X \cup Y$, $C = A \cup B$.

The fuzzy support value is calculated by multiplying the membership grade of each (z_j, c_j), summing them, then dividing the sum by the number of records [9]. We prefer the product operator as the fuzzy AND, instead of the normal minimum, because it better distinguishes high- and low-support transactions.

$$FS_{(Z,C)} = \frac{\sum_{i=1}^{n} \prod_{j=1}^{m} (t^i[(z_j, c_j)])}{n},$$

where m is the number of items in itemset (Z, C).

The fuzzy confidence value is calculated as follows:

$$FC_{((X,A),(Y,B))} = \frac{FS_{(Z,C)}}{FS_{(X,A)}}.$$

Both of the above formulas are direct generalizations of the corresponding formulas for the non-fuzzy case [1].

...	$(Age, middle)$	...	$(Income, low)$	...
	0.7		0.5	
	0.2		0.3	
...	0.5	...	0.2	...
	0.3		0.4	
	0.6		0.2	
	0.8		0.4	

Table 6: Part of a database containing fuzzy membership values

The following example illustrates the calculation of the fuzzy support and fuzzy confidence values. Let $Z = \{Age, Income\}$, $C = \{middle, low\}$ and a part of database shown in Table 6. The fuzzy support and confidence of (Z, C) are given by:

$$FS_{(Z,C)} = \frac{0.35+0.06+0.1+0.12+0.12+0.32}{6} = 0.178$$
$$FC_{((X,A),(Y,B))} = \frac{0.35+0.06+0.1+0.12+0.12+0.32}{0.7+0.2+0.5+0.3+0.6+0.8} = 0.345$$

3.4 Fuzzy Covariance and Correlation Values

Covariance is one of the simplest measures of dependence, based on the co-occurrence of the antecedent (X, A) and consequent (Y, B). If they co-occur clearly more often than what can be expected in an independent case, then the rule $(X, A) \rightarrow (Y, B)$ is potentially interesting. Piatetsky-Shapiro called this measure a *rule-interest* function [11]. We extend it to the fuzzy case, and define the covariance measure as:

$$FCov_{((X,A),(Y,B))} = FS_{(Z,C)} - FS_{(X,A)} \cdot FS_{(Y,B)}.$$

Covariance has generally the drawback that it does not take distributions into consideration. Therefore, in statistics, it is more common to use so called correlation measure, where this drawback has been eliminated. Again, we have to generalize the non-fuzzy formula to the fuzzy case, and obtain:

$$FCorr_{((X,A),(Y,B))} = \frac{FCov_{((X,A),(Y,B))}}{\sqrt{Var_{(X,A)} \cdot Var_{(Y,B)}}},$$

where

$$Var_{(X,A)} = FS_{(X,A)^2} - \left(FS_{(X,A)}\right)^2,$$
$$FS_{(X,A)^2} = \frac{\sum_{i=1}^{n} \left(\prod_{j=1}^{m} t^i\left[(x_j, a_j)\right]\right)^2}{n},$$

similarly for (Y, B).

These definitions are extensions of the basic formulas of variance and covariance. The value of the fuzzy correlation ranges from -1 to 1. Only a positive value tells that the antecedent and consequent are related. The higher the value is, the more related they are.

We use the information in Table 6 to illustrate the calculation of the fuzzy correlation value of a rule. Given the rule, "If *Age* is *middle* then *Salary* is *low*", the fuzzy covariance and correlation values of the rule are as follows:

$$FCov_{((X,A),(Y,B))} = 0.178 - 0.516 \cdot 0.333 = 0.006$$
$$FCorr_{((X,A),(Y,B))} = \frac{0.006}{\sqrt{0.045 \cdot 0.012}} = 0.258.$$

We defined the fuzzy extension of correlation measure, because it is an alternative to confidence, when measuring the dependence between the antecedent and consequent of a rule. We defined some other alternative measures of interestingness in [7]. In Section 5, we shall show results for both confidence and correlation.

4 Algorithm for Mining Fuzzy Quantitative Association Rules

An efficient algorithm for mining quantitative association rules has been proposed in [12]. However, a new algorithm is needed to solve the mining of fuzzy quantitative association rules. The problem of discovering all fuzzy quantitative association rules can be decomposed into two subproblems:

1. Find all itemsets that have fuzzy support $\left(FS_{(X,A)}\right)$ above the user specified minimum support (see Section 3.3). These itemsets are called *frequent itemsets.*

2. Use the frequent itemsets to generate the desired rules. The general idea is that if, say, X, Y, and $X \cup Y$ are frequent itemsets, then we can determine if the rule $X \Rightarrow Y$ holds by computing $FC_{((X,A),(Y,B))}$ (see Section 3.3). If this value is larger than the user specified minimum confidence value, then the rule will be interesting. We can also use the fuzzy correlation value $\left(FCorr_{((X,A),(Y,B))}\right)$ for this problem (see Section 3.4).

An algorithm for mining quantitative association rules has the following inputs an outputs.
Inputs: A database D, three threshold values *minsup*, *minconf* and *mincorr*.
Output: A list of interesting rules.
Notations:

D	the database
D^T	the transformed database
F_k	set of *frequent k-itemsets* (have k items)
C_k	set of *candidate k-itemsets* (have k items)
I	complete item set
minsup	support threshold
minconf	confidence threshold
mincorr	correlation threshold

Algorithm:
Main Algorithm $(minsup, minconf, mincorr, D)$
1 $F = \emptyset$;
2 $I = \text{Search}(D)$;
3 $(C_1, D^T) = \text{Transform}(D, I)$;
4 $k = 1$;
5 $F_k = \text{Checking}(C_k, D^T, minsup)$;
6 while $(|C_k| \neq 0)$ do
7 begin
8 $k = k + 1$;
9 if $k == 2$ then
10 $C_k = \text{Join1}(F_{k-1})$
11 else $C_k = \text{Join2}(F_{k-1})$;
12 $C_k = \text{Prune}(C_k)$;
13 $F_k = \text{Checking}(C_k, D^T, minsup)$;
14 $F = F \cup F_k$;
15 end
16 $Rules(F, minconf, mincorr)$;

The subroutines are outlined as follows:

1. **Search**(D): The subroutine accepts the database, finds out and returns the complete item set $I = \{i_1, i_2, \ldots, i_n\}$. For example, $I = \{Age, Income, Status\}$ for the database given in Table 1.

2. **Transform**(D, I): This step generates a new transformed (fuzzy) database D^T from the original database by user specified fuzzy sets. At the same time, the *candidate 1-itemsets* C_1 will be generated from the transformed database. (C_i is a set of sets of $(item, fuzzy\ set)$ pairs.) For example, $C_1 = \{\{(Age, young)\}, \{(Age, middle)\}, \{(Age, old)\}, \{(Income, low)\}, \{(Income, medium)\}, \{(Income, high)\}, \{(Status, unmarried)\}, \{(Status, married)\}\}$ is the complete set of *candidate 1-itemsets.*

3. **Checking**$(C_k, D^T, minsup)$: In this subroutine, the transformed (fuzzy) database is scanned and the fuzzy support $(FS_{(X,A)})$ of each candidate in C_k is calculated. A *k-itemset* in C_k is deleted if its fuzzy support is less than minsup. The remaining candidate itemsets will be kept in C_k. At the same time, the frequent itemsets F_k will be generated from C_k.

4. **Join1**(F_{k-1}): This Join step generates C_2 from F_1 as follows:
insert into C_2
select $\{(X, A), (Y, B)\}$
from $(X, A), (Y, B)$ in F_1
where $X \neq Y$

For example, after this Join step C_2 will be $C_2 = \{\{(Age, young),\ (Income, high)\}, \{(Age, middle), (Income, low)\}, \cdots\}$, but $C_2 \neq \{\cdots, \{(Age, young), (Age, middle)\}, \cdots\}$.

5. **Join2**(F_{k-1}): This Join step generates C_k from F_{k-1} as in [1]. For example, if we have $\{\{(Age, young), (Income, low)\}, \{(Age, young), (Balance, low)\}\}$ in F_{k-1}, $\{\{(Age, young),\ (Income, low), (Balance, low)\}\}$ will be generated in C_k.

6. **Prune**(C_k): During the prune step, an itemset S in C_k will be pruned if a subset of S does not exist in C_{k-1}.

7. **Rules**(F): Find the rules from the *frequent itemsets* F.

   For example, if $(Age, young)$ and $(Income, low)$ are frequent itemsets, then we get the $(Age, young) \Rightarrow (Income, low)$ rule, if its fuzzy confidence value (and fuzzy correlation value) is larger than the user specified minimum value.

5 Experimental Results

In this section, we will examine the accuracy and efficiency of our approach by experimenting with a real-life dataset. We applied our approach to a database called FAM95. This database contains data for the 63756 families that were interviewed in the March 1995 Current Population Survey (CPS), conducted by the Bureau of the Census for the Bureau of Labor Statistics. The data had 23 attributes: 7 quantitative and 16 categorical.

5.1 Interest Measures

In this experiment, we use six quantitative attributes to illustrate how the fuzzy concept gives more interesting rules than the discrete. The quantitative attributes were age of head in years ("head" is the reference person in a family), number of persons, children in family, education level of head, head's personal income and family income. Each quantitative attribute has three intervals/fuzzy sets. We choose the intervals by applying the well-known quantile-based partitioning, so that each interval gets the same number of attribute values.

Figure 4 shows the number of frequent itemsets for different minimum support. As expected, the number of frequent itemsets decreases as the minimum support increases from 10% to 45%. Fuzzy1 denotes the fuzzy method without normalization and Fuzzy2 denotes the fuzzy method with normalization. We can see that the fuzzy method with normalization gives fewer frequent itemsets than the fuzzy method without normalization. This method and the discrete interval method give similar numbers of frequent itemsets.

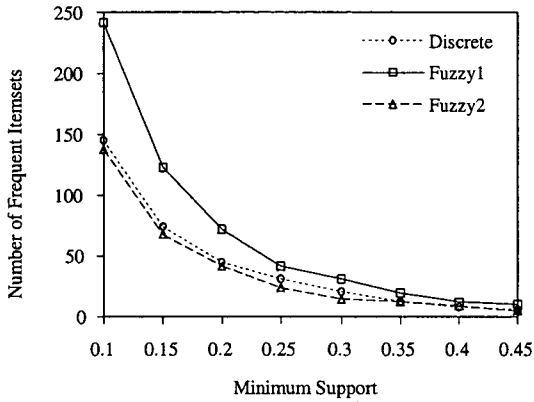


Figure 4: Number of frequent itemsets

Figures 5 and 6 show the number of interesting rules for different minimum confidence and correlation values. In both cases the minimum support was set to 20%. The results are quite similar to those of Figure 4.

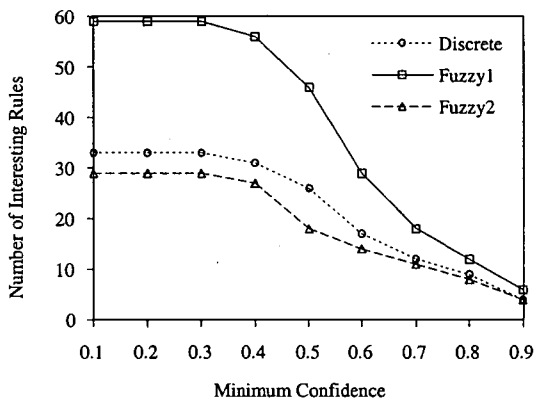


Figure 5: Effect of minimum confidence

We can see that the fuzzy method without normalization using confidence to calculate interest measure gives the highest number of expected interesting rules. However, in the correlation case the fuzzy method with normalization gives more rules than the others if the minimum correlation value was 0.6.

In the following we show some interesting rules. The minimum support was set

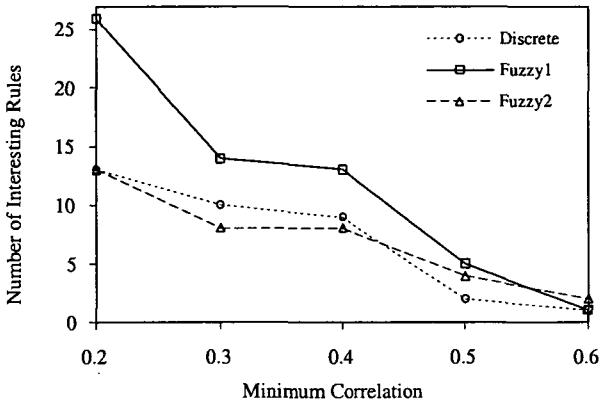


Figure 6: Effect of minimum correlation

to 20%, minimum confidence to 50%, and minimum correlation to 0.5.

IF *IncHead* is *low* THEN *IncFam* is *low*
IF *IncHead* is *medium* THEN *IncFam* is *medium*
IF *FamPers* is *low* AND *IncHead* is *low* THEN *IncFam* is *low*
IF *NumKids* is *low* AND *IncHead* is *low* THEN *IncFam* is *low*
IF *FamPers* is *low* AND *NumKids* is *low* AND *IncHead* is *low* THEN *IncFam* is *low*

5.2 Scale-Up Experiment

In this experiment, we will give the results on the performance of the algorithm using the confidence and correlation interest measures. The running time for the algorithm can be split into two parts:

- *Candidate generation.* The time for this is independent of the number of records.
- *Counting support, confidence and correlation.* The time for this is directly proportional to the number of records. When the number of records is large, this time will dominate the total time.

Thus we would expect the algorithm to have near-linear scaleup. This is confirmed by Figure 7, which shows the execution time as we increase the number of input records from 10000 to 64000. Note that we use five quantitative attributes in the database and each attribute has three fuzzy sets. We have set the user specified parameters such that both methods will give the same number of rules. The graph shows that the methods scale quite linearly for this dataset.

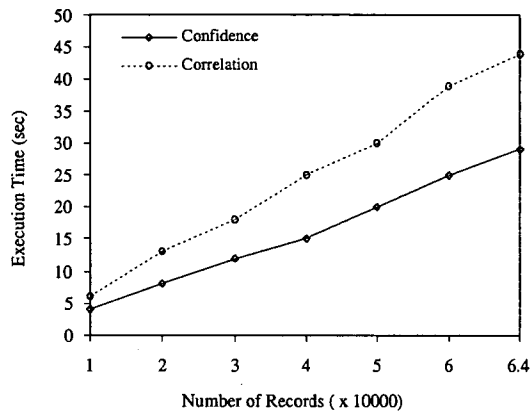


Figure 7: Scale-up: number of records

6 Conclusion

In this paper, we showed a new algorithm for mining fuzzy association rules, which were introduced in earlier papers. We assign each attribute with several fuzzy sets which characterize the quantitative attribute. Fuzzy sets provide a smooth transition between member and non-member of a set. We gave three different definitions for interest measure: fuzzy support, fuzzy confidence, and fuzzy correlation.

We showed two different methods of mining fuzzy quantitative association rules: without normalization, and with normalization. The unnormalized method gives the highest number of interesting rules. The normalized fuzzy method gives about the same number of rules as the discrete. However, either result set might not be included in the other.

We proposed a new algorithm for mining such quantitative association rules. Our experiments on a real-life dataset indicate that the algorithm scales linearly with the number of records. They also showed that the confidence interest measure gives better performance than the correlation interest measure.

Acknowledgment

I wish to thank Jukka Teuhola for his insightful comments and suggestions. Also thanks to the anonymous referee for promoting the clarity of the paper.

References

[1] Agrawal, R., Imielinski, T., Swami, A.: Mining association rules between sets of items in large databases. Proc. of ACM SIGMOD (1993) 207–216

[2] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. Proc. of the 20th VLDB Conference (1994) 487–499

[3] Chan, Keith C.C., Au, Wai-Ho: Mining Fuzzy Association Rules. Proc. of CIKM Conference, LasVegas, Nevada, USA (1997) 209–215

[4] Fayyad, U.M., Uthurusamy, R.: Efficient algorithms for discovering association rules. AAAI Workshop on KDD, Seattle, Washington, (1994) 181–192

[5] Gyenesei A. Data mining approach for solving decision support problems of warehouse networks. POLVAX (in Hungarian), (1999) 69–93.

[6] Gyenesei, A.: Determining Fuzzy Sets for Quantitative Attributes in Data Mining Problems. Proc. of Advances in Fuzzy Systems and Evol. Comp. (2001) 48–53

[7] Gyenesei A. Interestingness Measures for Fuzzy Association Rules. In Proc. of the 5th European Conference on PKDD (accepted) (2001)

[8] Hong, T-P., Kuo, C-S, Chi, S-C.: Mining association rules from quantitative data. Intelligent Data Analysis 3 (5) (1999) 363–376

[9] Kuok, C.M., Fu, A., Wong, M.H.: Fuzzy association rules in databases. In ACM SIGMOD Record 27(1),(1998) 41–46

[10] Park, J.S., Chen, M-S., Yu, P.S.: An effective hash-based algorithm for mining association rules. Proceedings of ACM SIGMOD, (1995) 175-186.

[11] Piatetsky-Shapiro, G., Frawley, W.J.: Knowledge Discovery in Databases. Chapter 13. AAAI Press/The MIT Press, Menlo Park, California (1991)

[12] Srikant, R., Agrawal, R.: Mining quantitative association rules in large relation tables. Proc. of ACM SIGMOD (1996) 1–12

[13] Srikant, R., Agrawal, R.: Fast algorithms for mining association rules. Proceedings of the 20th VLDB Conference (1994) 487–499.

CONTENTS

ISSN 0324—721 X

Felelős szerkesztő és kiadó: Csirik János